Command Line Interface Guide

Version 4.1

June 2000

ETNUS

Contents

1 TotalView Command Line Interpreter

What Is the CLI 1
Document Contents 2
Conventions 4
Reporting Problems 4

2 A Few CLI/Tcl Macros

Setting the EXECUTABLE_PATH State Variable 5
Print an Array Slice 6
Setting Breakpoints 8

3 Using the CLI

How a Debugger Operates 11
    Tcl and the CLI 12
    The CLI and TotalView 12
    The CLI Interface 13
Starting the CLI 14
    Debugger Initialization 15
    Executing a Start-Up File 15
    Starting Your Program 17
CLI Output 19
    "more" Processing 20
Built-In Aliases and Group Aliases 21
Command Arguments 22
Symbols 22
    Symbol Names and Scope 23
    Qualifying Symbol Names 24
Effects of Parallelism on TotalView and CLI Behavior 25
    Groups 26

Process/Thread Sets and Arenas 27
Specifying Processes and Threads 28
Incomplete Arena Specifiers 30
Lists With Inconsistent Widths 31
Kinds of IDs 31
Command and Prompt Formats 32
Controlling Program Execution 33
Advancing Program Execution 33
Action Points 35
Stepping 35

4 CLI Commands

Command Overview 37
alias 40
capture 42
dactions 43
dassign 45
dattach 47
dbreak 50
dcont 53
ddelete 55
ddetach 56
ddisable 57
ddown 58
denable 60
dfocus 61
dgo 63
dhalt 64
dkill 65
dlist 66
dload 70
dnext 72
dnexti 73
dprint 74
drerun 77
drun 78
dset 80
dstatus 85
dstep 86
dstepi 88

dunset 89
dup 90
dwait 92
dwatch 93
dwhat 96
dwhere 99
exit 101
help 102
quit 103
stty 104
unalias 105

A CLI Command Summary 107

B CLI Command Aliases and Focus 113

Glossary 117

Index 129

Chapter 1

TotalView Command Line Interpreter

This document describes the TotalView® Command Line Interpreter (CLI). The CLI is a command-oriented debugger that can be used as a stand-alone product or it can be used along with the TotalView Graphical User Interface (GUI) debugger. Depending upon your needs, you can view the CLI and TotalView debuggers as either being independent or complementary products. In most cases, the easiest way to debug programs is by using the TotalView GUI. However, there are circumstances when you need to do something not possible or practical using a GUI. For example, you may not want to interactively debug a program that takes days to execute.

The CLI debugger commands that you will execute are integrated within a Tcl interpreter. This combination removes the CLI from the realm of purely command-line debuggers in that you can add your own debugging commands, automate repetitive tasks, and even have the CLI run your program to a point where you are ready to begin debugging using the GUI. For example, you could ask the CLI to watch a memory location for changes and stop the program when a change occurs.

What Is the CLI

The CLI and TotalView are tools that give you visibility into, and control over, executable programs. An executable program has three basic components: the program's source files, its executables, and shared libraries. As the executable is running, it will also be using the stack and the heap.

The programs you will debug using the CLI are related to all three of these, but the component that is most often the target of CLI operations is either

an executing user program or a user program that you have loaded into memory.

The executing program has one or more *processes*, each associated with an executable (and perhaps one or more shared libraries) and each occupying a memory address space. Every process, in turn, has one or more *threads*, each with its own set of registers and its own stack.

The program being debugged is the complete set of threads and communicating processes that make up an application. The exact number of processes and threads depends on many factors, including how you wrote the program, the transformations performed by the compiler, the way the program was invoked, and the sequence of events that occur during execution. Thus, the number of processes and threads usually changes while your program executes.

Some operating systems, compilers, and run-time systems impose restrictions on the relationship between processes, threads, and executables. SPMD (Single Program Multiple Data) programs are parallel programs involving just one executable, executed by multiple threads and processes. MPMD (Multiple Program Multiple Data) programs involve multiple executables, each executed by one or more threads and processes.

Document Contents

Using the CLI assumes that you are familiar with and have experience debugging programs with TotalView. For example, this book does not explain how TotalView manipulates threads, processes, and groups. You will find that information in the TOTALVIEW USER'S GUIDE.

Similarly, CLI commands are embedded within Tcl, which means that you'll get the best results using the CLI if you are familiar with Tcl. Fortunately, you can find books describing Tcl at many book stores and you can also order these books from online bookstores. Two excellent books are

- Ousterhout, John K. Tcl and the Tk Toolkit. Reading, Mass.: Addison Wesley, 1997.
- Welch, Brent B. Practical Programming in Tcl & Tk. Upper Saddle River, N.J.: Prentice Hall PTR, 1997.

There is also a rich supply of resources available on the Web. Two starting points are **www.ajubasolutions.com** and **www.tcltk.com**.

The best way to understand the kinds of information in this book is to take a minute or two to browse through this book's table of contents. The fastest way to gain an appreciation of the actions performed by CLI commands is to review Appendix A, which contains an overview of CLI commands.

Here is how the information in this book is organized:

Chapter 1: TotalView Command Line Interpreter

This first chapter introduces the CLI.

Chapter 2: A Few CLI/Tcl Macros

Because you already know how to program, your biggest challenge in using the CLI will be remembering its commands and understanding how they are used within the Tcl environment. This chapter presents a few simple macros that demonstrate how the two are used together.

Chapter 3: Using the CLI

The CLI commands execute within the Tcl and TotalView environments. (The code used by the CLI and TotalView that interacts with your programs is shared.) This chapter explains how you specify processes, threads, and groups, name program locations, and the like.

Chapter 4: CLI Commands

This chapter contains the **man** pages for CLI commands.

Appendix A: CLI Command Summary

This appendix contains a listing of all CLI commands, a brief explanation for what the command does, and a depiction of the command's syntax.

Appendix B: CLI Command Aliases and Focus

Here you will find a table containing the predefined aliases for all commands and the default focus for each command. (The *focus* indicates the processes and threads upon which a command acts.)

Conventions

The following table describes the conventions used in this book:

TABLE 1: **Book Conventions**

Convention	Meaning
[ ]	Brackets are used when describing parts of a command that are optional. Be careful to distinguish between brackets used in command descriptions and brackets used by Tcl that evaluate expressions.
|	Choose one of the listed commands. (| means "or".)
arguments	Within a command description, text in italic represent information you type. Elsewhere, italic is used for emphasis. You will not have any problems distinguishing between the uses.
Dark text	Within a command description, **dark text** represent key words or options that you must type exactly as displayed. Elsewhere, it represents words that are used in a programmatic way rather than their normal way.

Reporting Problems

Please contact us if you have problems installing TotalView, questions that are not answered in the product documentation or on our Web site, or suggestions for new features or improvements.

Our Internet E-Mail addresses is **support@etnus.com**
1-800-856-3766 in the United States
(+1) 508-875-3030 worldwide

If you are reporting a problem, please include the following information:

- The **version** of TotalView and the **platform** on which you're running TotalView
- An **example** that illustrates the problem
- A **record** of the sequence of events that led to the problem

See the TOTALVIEW RELEASE NOTES for complete instructions on how to report problems.

Chapter 2

A Few CLI/Tcl Macros

You can use the CLI in two ways—and, of course, you can combine these two ways. The first is as a command-line debugger that acts as a complement to the TotalView Graphical User Interface (GUI) debugger. The second is as a debugging programming language that allows you to add your own commands and functions.

This chapter contains a few macros that show how the CLI programmatically interacts with your program and with TotalView. Reading a few examples without bothering too much with details will give you an appreciation for what the CLI can do and how it is used. As you will see, you really need to have a basic knowledge of Tcl before you can make full use of all CLI features.

Setting the EXECUTABLE_PATH State Variable

The following macro recursively descends through all directories starting at a location that you enter. (This is indicated by the *root* argument.) The macro will ignore directories named in the *filter* argument. The result is then set as the value of the CLI **EXECUTABLE_PATH** state variable.

```
#
#  Usage:
#
#   rpath <root> <filter>
#
# If <root> is not specified, start at the current directory.
#
# <filter> is a regular expression that removes unwanted
# entries. If it is not specified, the macro automatically filters
# out CVS/RCS/SCCS directories.
```

```
#
# The TotalView search path is set to the result.
#

proc rpath { { root "." } { filter "/(CVS|RCS|SCCS)(/|$)" } } {

    # Invoke the UNIX find command to recursively obtain all
    # directory names below “root”.
    set find [split [exec find $root -type d -print] \n]

    set npath ""

   # Filter out unwanted directories
    foreach path $find {
        if {! [regexp $filter $path] } {
            append npath $path:
        }
    }

    # Tell TotalView to use it
    dset EXECUTABLE_PATH $npath
}
```

In this macro, the final statement setting the **EXECUTABLE_PATH** state variable is the only uniquely CLI statement. The other statements are standard Tcl.

The **dset** command, like most CLI commands, begins with the letter **d**. (The **dset** command is only used when assigning values to CLI state variables. In contrast, values are assigned to Tcl variables using the standard Tcl **set** command.)

Print an Array Slice

The following macro prints a Fortran array slice. This macro, like other one shown in this chapter, relies heavily on Tcl and uses unique CLI commands sparingly.

```
proc pf2Dslice {anArray i1 i2 j1 j2 {i3 1} {j3 1} {width 20}} {
    for {set i $i1} {$i <= $i2} {incr i $i3} {
        set row_out ""
        for {set j $j1} {$j <= $j2} {incr j $j3} {
            set ij [capture dprint $anArray\($i,$j\)]
            set ij [string range $ij \
                [expr [string first "=" $ij] + 1] end]
            set ij [string trimright $ij]
            if { [string first "-" $ij] == 1} {
                set ij [string range $ij 1 end] }
            append ij "                    "
            append row_out " " [string range $ij 0 $width] " "
        }
        puts $row_out
    }
}
```

After invoking this macro, the CLI prints a two-dimensional slice **(i1:i2:i3, j1:j2:j3)** of a Fortran array to a numeric field whose width is specified by the **width** argument. (This width does not include leading minus sign.)

All but one line is standard Tcl. This line uses the **dprint** command to obtain the value of one array element. This element's value is then captured into a variable. (**dprint** does not return a value. The CLI **capture** command allows a value that is normally printed to be sent to a variable.)

Here are several examples.

```
d1.<> pf2Dslice a 1 4 1 4
   0.841470956802 0.909297406673 0.141120001673-0.756802499294
   0.909297406673-0.756802499294-0.279415488243 0.989358246326
   0.141120001673-0.279415488243 0.412118494510-0.536572933197
  -0.756802499294 0.989358246326-0.536572933197-0.287903308868
d1.<> pf2Dslice a 1 4 1 4 1 1 17
   0.841470956802 0.909297406673 0.141120001673-0.756802499294
   0.909297406673-0.756802499294-0.279415488243 0.989358246326
   0.141120001673-0.279415488243 0.412118494510-0.536572933197
  -0.756802499294 0.989358246326-0.536572933197-0.287903308868
d1.<> pf2Dslice a 1 4 1 4 2 2 10
   0.84147095 0.14112000
   0.14112000 0.41211849
d1.<> pf2Dslice a 2 4 2 4 2 2 10
  -0.75680249 0.98935824
   0.98935824-0.28790330
d1.<>
```

Setting Breakpoints

In many cases, your knowledge of what a program is doing lets you make predictions as to where problems will occur. The following CLI macro parses comments that you can include within a source file and, depending on the comment's text, sets a breakpoint or an evaluation point.

Immediately following this listing is an excerpt from a program that uses this macro.

```
# make_actions: Parse a source file, and insert
# evaluation and breakpoints according to comments.
#
proc make_actions { { filename "" } } {

    if { $filename == "" } {
        puts "You need to specify a filename"
        error "No filename"
    }

            # Open the program's source file and initialize a few
            # variables
    set fname [set filename]
    set fsource [open $fname r]
    set lineno 0
    set incomment 0

            # Look for "signals" that indicate the kind of action
            # point. These signals are buried in the C comments.
    while { [gets $fsource line] != -1} {
        incr lineno
        set bpline $lineno

                # Look for a one-line evaluation point. The
                # format is ... /* EVAL: some_text */.
                # The text after EVAL and before the "*/" in
                # the comment is assigned to "code".
        if [regexp "/\\* EVAL: *(.*)\\*/" $line all code] {
            dbreak $fname\#$bpline -e $code
            continue
        }
```

```
                # Look for a multiline evaluation point
            if [regexp "/\* EVAL: *(.*)" $line all code] {
                # Append lines to "code".
                while { [gets $fsource interiorline] != -1} {
                    incr lineno

                    # tabs will confuse dbreak.
                    regsub -all \t $interiorline "  " interiorline

                    # If "*/" is found, add the text to "code", then
                    # leave the loop. Otherwise, add the text, and
                    # continue looping.
                    if [regexp "(.*)\*/" $interiorline all interiorcode] {
                        append code \n $interiorcode
                        break
                    } else {
                        append code \n $interiorline
                    }
                }
                dbreak $fname\#$bpline -e $code
                continue
            }
                # Look for a breakpoint
            if [regexp "/\* STOP: .*" $line] {
                dbreak $fname\#$bpline
                continue
            }
                # Look for a command to be executed by Tcl.
            if [regexp "/\* *CMD: *(.*)\*/" $line all cmd] {
                puts "CMD: [set cmd]"
                eval $cmd
            }
        }
        close $fsource
    }
```

The only similarity between this example and the previous two is that almost all of the statements are Tcl. The only purely CLI commands are the instances of the **dbreak** command. (This command sets evaluation points and breakpoints.)

The following excerpt from a larger program shows how you would embed comments within a source file that would be read by this macro:

```
…
struct struct_bit_fields_only {
    unsigned f3 : 3;
    unsigned f4 : 4;
    unsigned f5 : 5;
    unsigned f20 : 20;
    unsigned f32 : 32;
} sbfo, *sbfop = &sbfo;
…
int main()
{
    struct struct_bit_fields_only *lbfop = &sbfo;
…
    int i;
    int j;

    sbfo.f3 = 3;
    sbfo.f4 = 4;
    sbfo.f5 = 5;
    sbfo.f20 = 20;
    sbfo.f32 = 32;
…
    /* TEST: Check to see if we can access all the values */
    i=i; /* STOP: // Should stop */
    i=1;/* EVAL: if (sbfo.f3 != 3) $stop; // Should not stop */
    i=2;/* EVAL: if (sbfo.f4 != 4) $stop; // Should not stop */
    i=3;/* EVAL: if (sbfo.f5 != 5) $stop; // Should not stop */
    …
    return 0;
}
```

The **make_actions** macro reads a source file one line at a time. As it reads these lines, the regular expressions look for comments that begin with **/* STOP**, **/* EVAL**, and **/* CMD**. After parsing the comment, it sets a breakpoint at a *stop* line, an evaluation points at an *eval* line, or executes a command at a *cmd* line.

Using evaluation points can be confusing because evaluation point syntax differs from that of Tcl. In this example, the **$stop** command is a command contained within TotalView (and the CLI). It is not a Tcl variable. In other cases, the evaluation statements will be in the C or Fortran programming languages.

Chapter 3

Using the CLI

The two components of the Command Line Interpreter (CLI) are the Tcl-based programming environment and the commands added to the Tcl interpreter that allow you to debug your program. This chapter looks at how these components interact and describes how you specify processes, groups, and threads.

This chapter tends to emphasize interactive use of the CLI rather than using the CLI as a programming language because many of the concepts that will be discussed are easier to understand in an interactive framework. However, everything that you will read can be used in both environments.

How a Debugger Operates

The CLI and TotalView debuggers affect the target program but are not part of the target program's process. That is, TotalView and the CLI run in separate processes, and their semantics are separate from the target program's semantics.

A CLI interaction has two kinds of input: the executables that make up the target program *and* the Tcl and CLI commands that you type or execute. Because the CLI debugs programs at the source code level, each executable must be associated with debugging information. On most systems, this requires that you create the executable with special compiler options. In almost all cases, you will use **–g**, which tells the compiler to add information that lets the CLI display high-level output to the user, expressed in terms of the procedures and variables used in the source code. This option also allows the CLI to access components of the program (such as source files), eliminating the need for some assistance from the user.

Tcl and the CLI

The TotalView CLI is built on top of version 8.0 of Tcl, and the TotalView CLI commands are built within the TotalView version of Tcl. The CLI is not a library of commands that you can bring into other implementations of Tcl. However, the Tcl integrated with the CLI supports all libraries and operations that run using version 8.0 of Tcl.

Integrating CLI commands into Tcl makes them intrinsic commands of Tcl. This means that you can enter and execute CLI commands in exactly the same way as you enter and execute built-in Tcl functions such as **file** or **array**. It also means that you can embed Tcl primitives and functions within CLI commands.

For example, you can create a Tcl list that contains a list of threads, use Tcl commands to manipulate that list, then have a CLI command operate on the elements of this list. Or, you create a Tcl function that dynamically builds the arguments that a process will use when it begins executing.

Because the CLI is an integral part of TotalView's version of Tcl, there are no differences between using a CLI command and using a Tcl command. Furthermore, all CLI operations can be manipulated by Tcl.

The CLI and TotalView

The following figure illustrates the relationship between the CLI, TotalView, and your program:

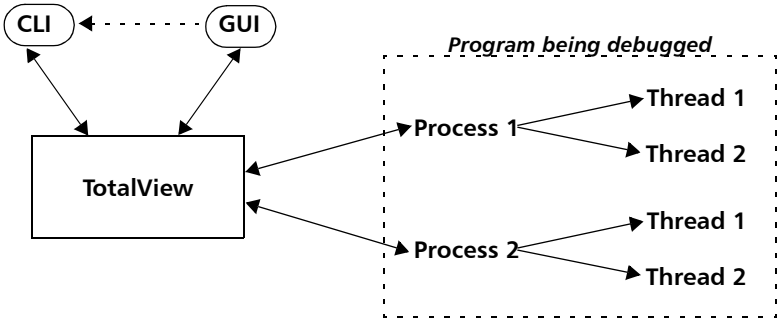

FIGURE 1: **The CLI and TotalView**

The CLI and the GUI are interfaces that communicate with TotalView, which is the component that actually performs the debugging work. The dotted arrow between the GUI and the CLI indicates that you can invoke the CLI from the GUI. The reverse is not true: you cannot invoke the GUI from the CLI.

In turn, TotalView communicates to the processes that make up your program and TotalView receives information back from these processes and passes them back to the component that sent a request.

The CLI Interface

The way in which you interact with the CLI is by entering a CLI command. Typically, the effect of executing a CLI command is one or more of the following:

- Information about the target program is displayed.
- A change takes place in the target program's state.
- A change takes place in the information that the CLI maintains about the target program.

The CLI signals that it has completed a command by displaying a prompt.

The CLI command performs actions sequentially, and it can only process one command at a time. For instance, if you enter a command that prints out an array, the CLI does not redisplay its prompt until all elements of the array are displayed.

Although CLI commands are executed sequentially, commands executed by your program may not be. For example, the CLI does not require that the target program be stopped when it prompts for and performs commands. It only requires that the last command be complete before it can begin executing the next one. In many cases, the processes and threads being debugged continue to execute while the CLI is performing commands.

Entering Ctrl-C while a CLI command is executing interrupts that CLI command or executing Tcl macro. If the CLI is displaying its prompt, typing Ctrl-C stops executing processes.

Starting the CLI

You can start the CLI in two ways:

- You can start the CLI from within the TotalView window by placing the cursor in the Root window, and then bringing up the TotalView pop-up menu. (In most cases, you will right-click your mouse button.) The menu displayed is as follows:

FIGURE 2: **TotalView Root Window Pop-up Menu**

After selecting the **Open Command Line Window** command, TotalView opens a window into which you can enter CLI commands. (The accelerator for this command is the capital letter "**C**".)

- You can start the CLI directly from a shell prompt by typing **totalviewcli**. (This assumes that the TotalView binary directory is in your path.)

Here is snapshot of a CLI window:

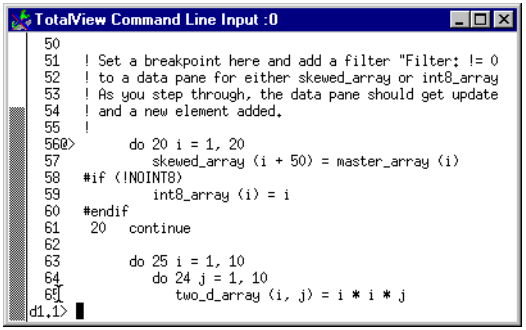

FIGURE 3: **CLI xterm Window**

If you have problems with command line editing, it could because you invoked the CLI from a shell or process that manipulates your **stty** settings. You can eliminate these problems if you use the **stty sane** CLI command. (If the **sane** option is not available, you will have to change values individually.)

Debugger Initialization

An *initialization file* contains commands that let you modify the CLI Tcl environment and add your own functions to this environment. This file, named **.tvdrc**, is read when the CLI begins executing. It can be located in your home directory or the directory from which you invoked TotalView. If it is present in both locations, the CLI executes both files. TotalView executes the file in your home directory before the file in the current directory.

Typically, the **.tvdrc** file contains command, function, and variable definitions and function calls that you want executed whenever you start a new debugging session.

Executing a Start-Up File

If you add the **-s** *filename* option to either the **totalview** or **totalviewcli** shell commands, you can have TotalView execute the CLI commands contained within *filename*.

The start-up file executes after **.tvdrc** files execute. The following figure shows the sequence in which initialization and startup files execute:

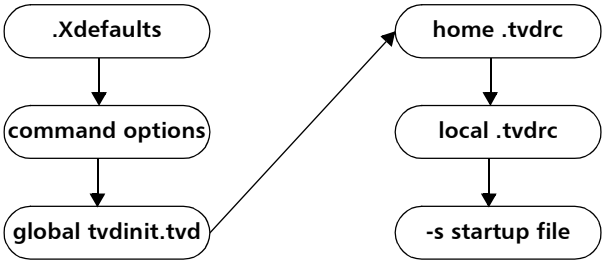

FIGURE 4: **Startup and Initialization Sequence**

This option lets you initialize the debugging state of your program, run the program you are debugging until it reaches some point where you are ready

to begin debugging, and even lets you create a shell command that starts the CLI.

Here is a small CLI program:

```
#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT { 0 4 -wp}
dstep
catch { make_actions fork_loop.cxx } msg
puts $msg
```

This program begins by loading and interpreting the **make_actions.tcl** file. (This was described in Chapter 2.) loads the **fork_loop** executable, sets its default start-up arguments, then steps one source level statement.

If this were stored in a file named **fork_loop.tvd**, here is how you would tell TotalView to start the CLI and execute this file:

```
totalviewcli -s fork_loop.tvd
```

Information on options and X Resources can be found in the TOTALVIEW USER'S GUIDE.

The following example shows how you would place a similar set of commands in a file that you would invoke from the shell:

```
#!/bin/sh
# Next line executed by shell, but ignored by Tcl because of: \
      exec totalviewcli -s "$0" "$@"
#
source make_actions.tcl
#
dload fork_loop
dset ARGS_DEFAULT {0 4 -wp}
dstep
catch {make_actions fork_loop.cxx} msg
puts $msg
```

Notice that the only difference is the first few lines in the file. In the second line, the continuation character is ignored by the shell. However, it is processed by Tcl. This means that the shell will execute it and Tcl will ignore it.

Starting Your Program

The CLI lets you start debugging operations in several ways. To execute the target program from within the CLI, enter a **dload** command followed by the **drun** command. The following example shows using the **totalviewcli** command to start the CLI. This is followed by **dload** and **drun** commands. As this was not the first time the file was run, breakpoints exist from a previous session.

NOTE In this listing, the CLI prompt is "d1.<>". The information preceding the final ">" symbol indicates the processes and threads upon which the current command act. Processes and threads are discussed throughout this chapter. In addition, the prompt itself is discussed in "Command and Prompt Formats" on page 32.

```
% totalviewcli
IRIX6 MIPS TotalView 4X.0.0-7
Copyright 1999 by Etnus, Inc. ALL RIGHTS RESERVED.
Copyright 1996-1998 by Dolphin Interconnect Solutions, Inc.
Copyright 1989-1996 by BBN Inc.

tcl_library is set to "/opt/totalview/lib"

d1.<> dload arrays              # load the "arrays" program
Mapping 430 bytes of ELF string data from 'arrays'...done
Digesting 42 ELF symbols from 'arrays'...done
Skimming 1825 bytes of DWARF '.debug_info' symbols from
    'arrays'...done
Indexing 408 bytes of DWARF '.debug_frame' symbols from
    'arrays'...done
...
Loading 1122 bytes of DWARF '.debug_info' information for
    arrays.F...done
1
d1.<> dactions                  # show action points
2 action points for process 1:
    1 addr=0x1000114c [arrays.F#53] Enabled
    2 addr=0x10000f34 [arrays.F#29] Enabled

d1.<> drun                      # run "arrays" until first action point
Created process 1/10715, named "arrays"
```

```
Thread 1.1 has appeared
d1.<> Thread 1.1 hit breakpoint 2 at line 29 in
    "check_fortran_arrays_"
```

This two-step operation allows you to set action points in the target program before execution begins. It also means that you can execute a program more than once, keeping TotalView state settings (such as the location of action points) in effect. At a later time, you can use the **drerun** command to tell the CLI to restart program execution, perhaps sending it new command-line arguments. In contrast, reentering the **dload** command tells the CLI to reload the program into memory (for example, after editing and recompiling the program). This has the side effect of creating new processes.

The **dkill** command forcibly terminates one or more processes of a program started by using **dload**, **drun**, or **drerun**. The following contrived examples continues executing the **arrays** program of the previous example:

```
d1.<> dkill                          # kill process
Process 1 has exited
d1.<> drun                           # runs "arrays" from beginning
Created process 1/10722, named "arrays"
Thread 1.1 has appeared
d1.<> Thread 1.1 hit breakpoint 2 at line 29 in "check_fortran_arrays_"
dlist -e -n 3                        #Shows lines about execution point
Loading 168 bytes of DWARF '.debug_info' information
for /comp2/mtibuild//targ64_m4/libftn/lseek64_.s...done
Loading 760 bytes of DWARF '.debug_info' information for
                              vtan.c...done
Loading 1864 bytes of DWARF '.debug_info' information for
    ns_passwd.c...done
    28                        do 10 i = 1, 100
    29@>                       master_array (i) = i * i * i
    30                        0   continue
d1.<> dwhat master_array     # Show me information
Loading 901 bytes of DWARF '.debug_info' information for
                              main.c...done
In thread 1.1:
Name: master_array; Type: integer*4(100); Size: 400 bytes;
                              Addr: 0xfffffac90
    Address class: auto_var (Local variable)Loading 188 bytes of
    DWARF '.debug_info' information for
/xlv55/irix/lib/libc/libc_64_M4/csu/crt1tinit.s...done
```

```
d1.<> drun                    # Notice the error message
drun: Process 1 already exists. Kill it first, or use rerun.
d1.<> dkill                   # kill processes again
Process 1 has exited

d1.<> drun
Created process 1/10730, named "arrays"
Thread 1.1 has appeared
d1.<> Thread 1.1 hit breakpoint 2 at line 29 in "check_fortran_arrays_"
```

In this example, notice that informational messages are interleaved (sometimes inconveniently) throughout the interaction. The occurs because the CLI prompt lets you know that the CLI is ready to accept another command. The messages, on the other hand, let you know what has happened with the executing process. These events seldom come at the same time.

Because information is interleaved, you may not realize that the prompt has appeared. It is always safe to use your Enter key to have the CLI redisplay its prompt. If a prompt is not displayed after you press Enter, then you know that the CLI is still executing.

NOTE To associate the CLI with a currently executing target program, use the dattach command.

CLI Output

A CLI command can either print its output to a window, or it can return the output as a character string. If the CLI executes a command that returns a string value, it also prints the returned string. Most of the time, you are not concerned with the difference between *printing* and *returning-and-printing*. Either way, information is displayed in your window. And, in both cases, printed output is fed through a simple *more* processor. (This is discussed in more detail in the next section.)

Here are two cases where it matters whether output is printed directly or is just returned:

- When the Tcl interpreter executes a list of commands, only the information returned from the last command is printed. Information returned by other commands is not shown.
- You can only assign the output of a command to a variable if the command's output is returned by the command. Output that is printed directly cannot be assigned to a variable or otherwise manipulated unless it is first saved with the **capture** command.

For example, the **dload** command returns the ID of the process object that was just created. The ID is normally printed by the interpreter—unless, of course, the **dload** command appears in the middle of a list of commands. For example:

```
{ dload test_program ; dstatus }
```

In this case, the CLI does not display the ID of the loaded program since **dload** was not the last command in the list. On the other hand, you can easily assign the ID of the new process to a variable:

```
set pid [dload test_program]
```

In contrast, you cannot assign the output of the **help** command to a variable. For example, the following does not work:

```
set htext [help]
```

This statement assigns an empty string to **htext** because **help** does not return text; it just prints information.

To capture the output of a command that normally prints its output, use the **capture** command. For example, the following places the output of the **help** command into a variable:

```
set htext [capture help]
```

"more" Processing

When the CLI displays output, it sends data through a simple internal *more*-like process. This process prevents data from scrolling off the screen before it can be viewed. After you see the **MORE** prompt, you must enter a Return to continue with the next screen of data. If you type **q** (followed by a Return), any remaining buffered output is discarded.

You can control the number of lines displayed between prompts by setting the **LINES_PER_SCREEN** state variable. (See **dset** on page 80 for more information.)

Built-In Aliases and Group Aliases

Almost every CLI command has an alias that allows you to abbreviate the command's name. (An alias is one or more characters that the Tcl interprets as a command.) These aliases are listed in Appendix B.

NOTE The alias command (see Chapter 4) lets you create your own aliases.

After a few minutes of entering CLI commands, you will quickly come to the conclusion that is much more convenient to use the command abbreviation. For example, you could type:

```
dfocus g dhalt
```

This command tells the CLI to halt the current group. However, it is much easier to type:

```
f g h
```

While less-used commands are often typed in full, a few commands are almost always abbreviated. These command include **dbreak** (**b**), **ddown** (**d**), **dfocus** (**f**), **dgo** (**g**), **dlist** (**l**), **dnext** (**n**), **dprint** (**p**), **dstep** (**s**), and **dup** (**u**).

The CLI also includes uppercase "group" versions of aliases for a number of commands, including all stepping commands. For example, the alias for **dstep** is "**s**"; in contrast, "**S**" is the alias for "**dfocus g dstep**". These group aliases differ from "native" group-level commands in two ways:

- They do not work if the current focus is a list. The **g** focus specifier modifies the current focus, and it can only be applied if the focus contains just one term.
- They always act on the group, no matter what width is specified in the current focus. Therefore, **dfocus t S** does a step-group command.

Command Arguments

The default command arguments for a process are stored in the **ARGS(#)** variable, where **#** is the CLI ID for the process. If the **ARGS(#)** variable is not set for a process, the CLI uses the value stored in the **ARGS_DEFAULT** variable. **ARGS_DEFAULT** is set if you had used the **-a** option when starting the CLI or TotalView. For example:

```
totalviewcli -a argument-1, argument-2 , ...
```

NOTE **The -a option tells TotalView to pass the information that follows to the program.**

To set (or clear) the default arguments for a process, you can use **dset** to modify the **ARGS()** variables directly, or you can start the process with the **drun** command. For example, here is how you can clear the default argument list for process 2:

```
dunset ARGS(2)
```

The next time process 2 is started, the CLI uses the arguments contained within **ARGS_DEFAULT**.

You can also use the **dunset** command to clear the **ARGS_DEFAULT** variable. For example:

```
dunset ARGS_DEFAULT
```

All commands (except **drun**) that create a process—including **dgo**, **drerun**, **dcont**, **dstep**, and **dnext**—pass the default arguments to the new process. The **drun** command differs in that it replaces the default arguments for the process with the arguments that are passed to it.

Symbols

This section discusses how the CLI handles *symbols* and other names corresponding to various entities within the program state, the machine state, or the TotalView state.

Symbol Names and Scope

Many commands refer to one or more program objects by using symbol names as arguments. In addition, some commands take expressions as arguments, where the expression can contain symbol names representing program variables.

NOTE Because the CLI is built on top of TotalView, the way in which the CLI interprets symbols is the way that TotalView interprets them.

TotalView learns about a program's symbols and their relationships by reading the debugging information that was generated when the program was compiled. The information includes a mapping from symbol names to descriptions of objects, providing information about a symbol's use (for example, a function), where it is located in memory after the executable is loaded, and associated features (for example, number and data types of a function's arguments). While TotalView smooths over many differences, the information provided by compiler manufacturers is not uniform, and differences exist between the kinds of information provided by Fortran, C, and C++ compilers.

In all cases, the concept of scope is central to the way TotalView interprets and accesses symbols. For the languages debugged by TotalView, a program consists of one or more scopes that are established by the program's structure. Typically, some scopes are nested within others. Every statement in a program is associated with a particular scope, and indirectly with the other scopes containing that scope.

Whenever a CLI command contains a symbol name, TotalView consults its debugging information to discover what object it refers to—this process is known as *symbol lookup*. Symbol names are not required to be unique within a program, making the task of symbol lookup both complex and context–sensitive. A symbol lookup is performed with respect to a particular context, expressed in terms of a single thread of execution. Each context uniquely identifies the scope to which a symbol name refers.

Qualifying Symbol Names

While the commands **dup** and **ddown** (which move up and down the stack) let you change the current context—which changes the target scope being searched—to some frame other than the one currently executing, standard lookup methods are cumbersome when you are examining a symbol in some other scope.

The syntax for qualifying a symbol with a scope closely resembles that for specifying a source location. The scopes within a target program form a tree, with the outermost scope as the root. At the next level are executable files and dynamic libraries; further down are compilation units (source files), procedures, and other scoping units (for example, blocks) supported by the programming language. Qualifying a symbol is equivalent to specifying which scope it is in, or describing the path to a node in the tree. This is similar to describing the path to a file in a tree-structured file system.

A symbol is *fully qualified* in terms of its scope when all levels of the tree are included:

[**#***executable-or-lib***#**][*file***#**][*procedure-or-line***#**]*symbol*

In this definition, the pound sign (**#**) is a separator character.

The components of the symbol name are interpreted as follows:

- Just as file names need not be qualified with a full path, you can qualify a symbol's scope without including all levels in the tree.
- Because programming languages typically do not let you name blocks, that portion of the qualifier is specified as a line number within the block.
- If a qualified symbol begins with **#**, the name that follows indicates the name of the executable or shared library (just as an absolute file path begins with a directory immediately within the root directory). If the executable or library component is omitted, the qualified symbol does not begin with **#**.
- The source file's name may appear after the (possibly omitted) executable or shared library.
- The procedure name or block component (represented by a line number from that block) may appear after the (possibly omitted) source file name. This component is followed by **#**.

- The symbol name follows the (possibly omitted) procedure or block name. Since qualified symbols often appear in the context of an expression, the final symbol name could be followed by a dot (.), plus the name of a field from a class, union, or structure.

You can omit any part of the scope specification that is not needed to uniquely identify the symbol. Thus, **foo#x** identifies the symbol **x** in the procedure **foo**. In contrast, **#foo#x** identifies either procedure **x** in executable **foo** or variable **x** in a scope from that executable.

Similarly, **#foo#bar#x** identifies variable **x** in procedure **bar** in executable **foo**. If **bar** were not unique within that executable, the name would be ambiguous unless you further qualified it by providing a file name. Ambiguities can also occur if a file-level variable (common in C programs) has the same name as variables declared within functions in that file. For instance, **bar.c#x** refers to a file-level variable, but the name can be ambiguous when there are different definitions of **x** embedded in functions occurring in the same file. In this case, you would need to say **bar.c#1#x** to identify the scope that corresponds to the "outer level" of the file (that is, the scope containing line 1 of the file).

You can use the **dwhat** command to determine if an unqualified or partially qualified symbol name is ambiguous.

Effects of Parallelism on TotalView and CLI Behavior

A parallel program consists of some number of processes, each involving some number of threads. Processes fall into two categories, depending on when they are created:

- **Initial process**

  A preexisting process from the normal run-time environment (that is, created outside the debugger) or one that was created as TotalView loaded the target program.

- **Spawned process**

  A new process created by a process executing under the CLI's control.

TotalView assigns an integer value to each individual process and thread under its control. This *process/thread identifier* can be the system identifier associated with the process or thread. However, it can be an arbitrary value created by the CLI. Process numbers are unique over the lifetime of a debugging session; similarly, thread numbers are also unique over the lifetime of a process.

Process/thread notation lets you identify the component that a command targets. For example, if your program has two processes, and each has two threads, four threads exist:

Thread 1 of process 1
Thread 2 of process 1
Thread 1 of process 2
Thread 2 of process 2

You would identify the four threads as follows:

1.1—Thread 1 of process 1
1.2—Thread 2 of process 1
2.1—Thread 1 of process 2
2.2—Thread 2 of process 2

Groups

When you start a multiprocess program, the CLI adds each process to a process group as the process starts. The debugger groups the processes depending on the type of system call (**fork()** or **execve()**) that created or changed the processes. There are two different types of process groups:

Program Group — Includes the parent process and all related processes. A program group includes children that were forked (processes that share the same source code as the parent) and children that were forked but which subsequently called **execve()**. That is, these are processes that do not share the same source code as the parent.

Share Group — Includes only the related processes that share the same source code.

In general, if you are debugging a multiprocess program, the program group and share group differ only when the program has children that are forked with a call to **execve()**.

Process/Thread Sets and Arenas

A P/T (P*rocess/*T*hread*) *set* is a list of one or more of the threads known to the CLI. You create a P/T set by placing one or more P/T identifiers within a Tcl list; that is, you enter them within braces (**{}**) or use Tcl commands that create and manipulate lists, or, in some cases, enter them as an argument to a CLI command. For example, the following list contains specifiers for process 2, thread 1 and process 3, thread 2:

```
{ p2.1 p3.2 }
```

NOTE A P/T list element is either a single thread or an "arena" of threads, as is explained below.

Unlike a serial debugger, where each command clearly applies to the only executing thread, the CLI usually controls and monitors many threads and many different locations corresponding to a program symbol name (for example, a variable). The concept of the *target* P/T *set* restricts a CLI command so that it applies to one, many, or all threads of control.

In you do not explicitly specify a P/T set, the target set is implicitly defined. This set is displayed as the (default) CLI prompt. You can change the set to which a command is applied with the **dfocus** command. If **dfocus** is executed as a separate command, it changes the default P/T set. If, however, it is contained as part of another command, it just changes the target of this command. After the command executes, the *old* default is restored. For example, assume that the current focus is process 1, thread 1. The following commands change the focus to group 2 and then steps the group twice:

```
dfocus g2
dstep
dstep
```

In contrast, the following commands step group 2, then steps process 1, thread 1:

```
dfocus g2 dstep
dstep
```

NOTE This chapter almost always presents the long, unabbreviated name for CLI commands. When you are interactively using the CLI, you would probably use the predefined aliases. For example, it is much easier to type "f g2 s" instead of "dfocus g2 dstep". (See Appendix B for a list of aliases.)

Some commands can be applied only at the process level—that is, you cannot apply them to a single thread (or group of threads) in the process, but must apply them to all or to none.

Specifying Processes and Threads

The P/T set is specified as a list of arena specifiers where an *arena* is the processes, threads, and groups that are affected by a CLI debugging command. Each *arena specifier* describes a single area in which a command will act; the *list* is just a collection of arenas. Most commands iterate over the list, acting on each arena in turn. Some output commands, however, may combine the arenas and act on them as a single target.

An arena specifier includes a *width specifier* and a *thread of interest*. ("Width specifiers" are discussed later in this section.)

The thread of interest is specified as **p.t** where **p** is the TotalView process ID (PID) and **t** is the TotalView thread ID (TID).

In addition, the less-than symbol (**<**) character, when used in place of the TID, indicates the *first user thread* in the process. This is useful when the first user thread is not thread 1; for example, the first and only thread may be thread number 3 on Compaq systems.

The complete P/T set specifier has the following form:

```
<width><process_id>.<thread_id>
```

width indicates how large a set of processes and threads are affected by each command, and is one of the following letters:

t *Thread width*

Just one thread is the target of the action.

p *Process width*

The process containing the thread of interest is the target of the action.

g *Group width*

The group containing the process of the thread of interest is the target of the action.

a *All processes*

Every process known to the CLI is the target of the action.

d *Default width*

The width of action will depend on the default for each command. This is the width to which the default focus is set. For example, the **dstep** command defaults to process width (run the process while stepping one thread), and the **dwhere** command defaults to thread width (backtrace just one thread).

The **process_id.thread_id** combination identifies the process and thread of interest. The thread of interest is the primary thread that is affected by a command. For example, the **dstep** command always steps the thread of interest, but it may optionally run the rest of the threads in the process of interest and may step other processes in the group.

NOTE On some systems, TotalView cannot distinguish manager threads from user threads, and manager threads may be chosen by mistake.

The thread of interest specifies a particular target thread, while the width specifies how many threads surrounding the thread of interest are affected. For example, the **dstep** command always requires a thread of interest, but entering this command can

- Run just the thread of interest during the step operation (single-thread single-step).
- Run all threads in the containing process during the operation (process-level single-step).
- Step all processes in the group which have threads at the same PC (program counter) as the thread of interest (group-level single-step).

Here are some examples:

d1.<	Use the default set for each command, focusing on the first user thread in process 1.
g2.3	Select process 2, thread 3; commands act on the entire group.
t1.7	Commands act only on thread 7 of process 1.

You can leave out parts of the P/T set if what you do enter is unambiguous. A missing width or PID is filled in from the current focus. A missing TID is always assumed to be <.

To save a P/T set definition for later use, assign it to a Tcl variable. For example:

```
set myset { g2.3 t3.1 }
dfocus $myset dgo
```

The thread of interest can also be modified by a *width* specifier.

Incomplete Arena Specifiers

In general, you do not need to completely specify an arena. Missing components are assigned default values or are filled in from the current focus. The only requirement is that the meaning of each part of the specifier cannot be ambiguous. Here is how the CLI fills in missing pieces:

- A dot typed before or after the number lets the CLI know if you are specifying a process or a thread. For example, "**1.**" is clearly a PID, while "**.7**" is clearly a TID.
- If you enter a width without specifying a process or a thread, the CLI uses the PID and TID from the current focus.
- If the width is **p** or **t**, you can omit the dot.

  If you use a process or thread width specifier, and give just one number, the CLI assumes that the number is either a process or a thread ID. For instance, **p3** refers to process 3, while **t7** refers to thread 7.
- If you do not use a width, the CLI uses the width from the current focus.
- If you do not use a PID, the CLI uses the PID from the current focus.
- If you use a PID and do not specify a TID, the CLI assumes that you are specifying TID < where < indicates the first thread.

  In most cases, the CLI does not use the TID from the current focus, since the TID is a process-relative value. Note that this is an important case for

handling groups of single-threaded processes or groups of processes with just one user thread (which should be thread 1).

- If you set the focus to a list, no focus-based defaulting is allowed.

  Using lists as a focus requires that you also use an explicit width and PID. In this case, the CLI will not look for default values from the current focus. If you omit the thread, the CLI uses the first thread; that is the thread obtained when you use <.

Lists With Inconsistent Widths

All arena specifiers have the same format. This can be confusing when a list contains more than one width specifier. Consider the following:

```
{ p2 g3.4 t7 }
```

This set of objects is clear enough: all of process 2, all processes in the same group as process 3, thread 4, and thread 7. However, how should the CLI use this set of processes, groups, and threads?

A simplistic answer is that the command uses the width of each of these arenas to determine the number of threads from the arena on which it will act, and then act on these threads. This is exactly what the **dgo** command does. In contrast, the **dwhere** command creates a call graph for process-level arenas, and the **dstep** command runs all threads in the arena while stepping the thread of interest. It may wait for threads in multiple processes for group-level arenas.

More generally, when the focus is a list, most commands iterate over the list and act on each arena. If the CLI cannot interpret an inconsistent focus, it prints an error message.

Kinds of IDs

In a multithreaded, multiprocess, distributed program, you are exposed to a many kinds of IDs. Here is some background on the kinds used in the CLI and TotalView:

System pid — This is the process ID and is generally called the *pid*. This ID is usually has a value between 100 and 32,000. However, it can be higher on some systems.

Debugger pid This is a sequentially numbered value beginning at 1 that is incremented for each new process. These IDs are used in the CLI to identify processes. Note that if the target process is killed and restarted (that is, you use the following two commands: **dkill** and **drun**), the debugger pid will remain the same. The system pid, however, changes since this is a new target process.

System tid This is the ID of the target system kernel or user thread. On some systems, the target thread IDs have no obvious meaning (for example, AIX). On other systems, they start at 1 and are incremented by 1 for each process.

TotalView thread ID This is usually identical to the system thread ID. On some systems (such as AIX where the threads have no obvious meaning), TotalView uses its own IDs.

Pthread ID This is the ID assigned by the Posix PTHREADS package. On most systems, this differs from the system thread ID. In these cases, it is a pointer value that points to the Pthread ID.

Command and Prompt Formats

The appearance of the CLI prompt lets you know that the CLI is ready to accept a user command. By default, the prompt lists the current focus, and then displays a greater than symbol (>) and a blank space. (The *current focus* is the processes and threads to which the next command applies.) For example (and these are the same examples used in a previous section of this chapter):

d1.<> The current focus is the default set for each command, focusing on the first user thread in process 1.

g2.3> The current focus is process 2, thread 3; commands act on the entire group.

t1.7> The current focus is thread 7 of process 1.

You can change the prompt's appearance by using the **dset** command to set the **PROMPT** state variable. For example:

```
dset PROMPT "Kill this bug!"
```

Controlling Program Execution

Execution control is relatively simple in a serial debugging environment. The target program is either stopped or running. When the program is running, an event such as arriving at a breakpoint can occur, which tells the CLI to stop the program. Sometime later, you will probably tell the serial program to continue executing. Parallel program execution is more complex, however, since each thread has an individual execution state. When a thread (or set of threads) triggers a breakpoint, a question arises as to what, if anything, should be done about the other threads and processes.

The processes and threads that the CLI will stop depends on whether the breakpoint is marked as stopping the group or the process. If a group is stopped, all processes in the program group are stopped.

The way in which the CLI tells processes and threads to resume execution (that is, you enter a **dgo**, **dstep**, **dcont**, or **dwait** command) is exactly opposite to the way it stops processes and threads: a resume command *undoes* the effect of triggering a breakpoint.

Advancing Program Execution

Debugging begins by executing a **dload** or **dattach** command. If **dload** is used, it must be followed by **drun**. These three commands apply to entire processes, not to individual threads. (The same is true of the **dkill** command.)

To advance program execution, you enter a command that causes one or more threads to execute instructions. The commands are applied to a target P/T set, making it possible to advance certain processes while others are held back. You can also advance program execution by increments, *stepping* the target program forward, and you can define the size of the increment.

In most debugging sessions, you are telling the CLI to stop and start your program. After a process stops, you can examine its state. In this sense, a debugger can be thought of as tool that allows you to alter a program's state

in a controlled way. And, debugging is the process of stopping the process to examine its state. However, the term "stop" has a slightly different meaning in a multiprocess, multithreaded program; it now means that the CLI holds one or more threads at a fixed execution location until you enter a command that tells them to resume their execution.

Once a program is loaded, each process that is actively executing moves among three execution states: running, stopped/runnable, or stopped/held.

Running State: The *running state*, which means that one or more processes or threads are running, is defined from the perspective of the CLI. From the perspective of the underlying run-time environment, a process may make many transitions between being ready to run and actually running, but these lower-level transitions are invisible to TotalView and the CLI.

Stopped/Runnable State: A process enters the *stopped/runnable state* when any of the following occurs:

- The executable is first loaded or TotalView first attaches to an existing process.
- You explicitly tell the CLI to stop the process.
- The process's execution triggers a program event.
- Some other process's execution triggers a program event that affects this thread.

Once the process stops, you can use CLI commands to examine and change the process's state. After performing some operation, you will usually tell it to resume executing.

Stopped/Held State: The *stopped/held state* is similar to stopped/runnable, differing in that a process in this state does not respond to resume commands. A process typically enters this state as a result of hitting a barrier and waiting for the remaining threads to enter this same barrier. The process's state must first be changed to stopped/runnable (when the barrier has been satisfied or by an explicit user command) before it is eligible for resuming.

Action Points

By defining *action points*, you can tell the CLI that it should stop a program's execution. The CLI lets you specify four different kinds of action points:

- A *breakpoint* stops the process when the program reaches a location in the source code.
- A *watchpoint* stops the process when the value of a variable is changed.
- A *barrier point*, as its name suggests, effectively prevents processes from proceeding beyond a point until other processes arrive. This gives you a method for synchronizing the activities of processes. (Note that barriers can only be applied to entire processes, not to individual threads.)
- An *evaluation point* lets you programmatically evaluate the state of the process or variable when execution reaches a location in the source code. Evaluation points typically do not stop the process; instead, they perform an action.

NOTE Extensive information on action points can be found in the TotalView User's Guide.

Each action point is associated with an *action point identifier*. You use these identifiers when you need to refer to the action point. Like process and thread identifiers, action point identifiers are assigned numbers as they are created. The ID of the first action point created is 1. The second ID is 2, and so on. These numbers are never reused during a debugging session.

The CLI and TotalView only let you assign one action point to a source code line. However, neither limits the complexity of an action point.

Stepping

TotalView can step the target by machine instructions or source statements. If the focus is set to a list with more than one term, all step commands iterate over the list, applying their actions to each term.

The action taken on each arena in the focus list depends on whether the term is a thread, process, or group width.

- **Thread**

  The thread of interest is stopped. Stepping a thread is not the same as stepping the thread's process. Note that not all systems can step threads. A width of **t** tells the CLI that it should just run that thread. In contrast, a process width (**p**) tells the CLI that it should run all threads in the process that are allowed to run while the thread of interest is stepped.

  TotalView also allows all manager threads to run.

- **Process** (default)

  The CLI allows the entire process to run, and execution continues until the thread of interest arrives at its goal location (the next statement, the next instruction, and such.). It plants a temporary breakpoint at the goal location for the duration of the command. If another thread reaches the goal breakpoint first, that thread steps over the temporary breakpoint, and the process is continued again, until the thread of interest reaches the goal.

- **Group**

  The CLI examines the group and identifies each process having a thread stopped at the same location as the thread of interest (a "matching" process). TotalView waits until the thread of interest and one thread from each matching process have all arrived at the action point.

In all cases, if the focus process does not exist before the command is executed, TotalView creates the process and then executes the command. The default arguments for the process are passed to it.

Chapter 4

CLI Commands

This chapter contains detailed descriptions of CLI commands.

Command Overview

This section lists all of the CLI commands. It also contains a short explanation of what each command does.

NOTE The parentheses following a command's name contains the command's alias. The command aliases in uppercase letters apply to groups. Not all commands have aliases.

Environment Commands

The CLI commands in this group provide information on the general CLI operating environment:

- *alias*: creates or views user-defined commands.
- *capture*: allows commands that print information to instead send their output to a variable.
- *dset*: changes or views values of CLI state variables.
- *dunset*: restores default settings of CLI state variables.
- *help (he)*: displays help information.
- *stty*: sets terminal properties.
- *unalias*: removes a previously defined command.

CLI Initialization and Termination

These commands initialize and terminate the CLI session, and add processes to CLI control:

- *dattach (at)*: brings one or more processes currently executing in the normal run-time environment (that is, outside the CLI) under CLI control.
- *ddetach (det)*: detaches the CLI from a process.
- *dkill (k)*: kills existing user processes, leaving debugging information in place.
- *dload (lo)*: loads debugging information about the target program into TotalView and prepares it for execution.
- *drerun (rr)*: restarts a process.
- *drun (r)*: starts or restarts the execution of user processes under control of the CLI.
- *dstatus (st, ST)*: shows current status of processes and threads.
- *exit, quit*: exits from the CLI, ending the debugging session.

Program Information

The following commands provide information about a program's current execution location, as well as allow you to browse the program's source files:

- *dassign (as)*: changes the value of a scalar variable.
- *ddown (d)*: navigates through the call stack by manipulating the current frame.
- *dlist (l)*: browses source code relative to a particular file, procedure, or line.
- *dprint (p)*: evaluates an expression or program variable and displays the resulting value.
- *dup (u)*: navigates through the call stack by manipulating the current frame.
- *dwhat (wh)*: determines what a name refers to.
- *dwhere (w)*: prints information about the target thread's stack.

Execution Control

The following commands control execution:

- *dcont* (*co*, CO): continues execution of processes and waits for them.
- *dfocus* (*f*): changes the set of process, threads, or groups upon which a CLI command acts.
- *dgo* (*g*, G): resumes execution of processes (without blocking).
- *dhalt* (*h*, H): suspends execution of processes.
- *dnext* (*n*, N): executes statements, stepping over subfunctions.
- *dnexti* (*ni*, NI): executes machine instructions, stepping over subfunctions.
- *dstep* (*s*, S): executes statements, moving into subfunctions if required.
- *dstepi* (*si*, SI): executes machine instructions, moving into subfunctions if required.
- *dwait*: blocks command input until processes stop.

Action Points

The following action point commands are responsible for defining and manipulating the points at which the flow of program execution should stop so that you can examine debugger or program state:

- *dactions* (*ac*): views information on action point definitions and their current status.
- *dbreak* (*b*): defines a breakpoint.
- *ddelete* (*de*): deletes an action point.
- *ddisable* (*di*): temporarily disables an action point.
- *denable* (*en*): reenables an action point that has been disabled.
- *dwatch* (*wa*): defines a watchpoint.

alias

Creates or views user-defined commands

Format:

Creates a new user-defined command

alias *alias-name defn-body*

Views previously defined commands

alias [*alias-name*]

Arguments:

alias-name	The name of the command or command argument being defined.
defn-body	The text that Tcl will substitute when it encounters *alias-name*.

Description:

The **alias** command associates a name you specify with one or more Tcl and TotalView commands. After you create an alias, you can use it in the same way as a native TotalView or Tcl command. In addition, you can include an alias as part of a definition of another alias.

If you do not enter an *alias-name* argument, the CLI displays the names and definitions of all aliases. If you specify an *alias-name* argument, the CLI displays the definition of the alias.

Because the **alias** command can contain Tcl commands, you must ensure that *defn-body* complies with all Tcl expansion, substitution, and quoting rules.

TotalView's global start-up file, **tvdinit.tvd**, defines a set of default aliases. All the common commands have one- or two-letter aliases. (You can obtain a list of these commands by typing **alias**—being sure not to use an argument—in the CLI window.)

You cannot use an alias to redefine the name of a CLI-defined command. You can, however, redefine a built-in CLI command by creating your own Tcl procedure. For example, here is a procedure that disables the built-in **dwatch** command. When a user types **dwatch**, this code will be executed instead of the built-in CLI code:

```
proc dwatch {} {
    puts "The dwatch command is disabled"
}
```

The CLI does not parse *defn-body* (the command's definition) until it is used. Thus, you can create aliases that are nonsensical or incorrect. Errors are only detected when Tcl tries to execute your command.

When you obtain help for a command, the help text includes the command's alias.

Examples:

alias nt dnext Defines a command called **nt** that executes the **dnext** command.

alias nt Displays the definition of the **nt** alias.

alias Displays the definitions of all user-defined commands.

alias m {dlist main}
Defines a command called **m** that lists the source code of function **main**.

alias step2 {dstep; dstep}
Defines a command called **step2** that does two **dstep** commands in a row. This new command will apply to the focus current when the command is issued.

alias step2 { s ; s} Creates an alias that performs the same operations as the one in the previous example. It differs in that it uses the alias for **dstep**. Note that you could also create an alias that does the same thing as follows: **alias step2 { s 2 }**.

alias step1 {f p1. dstep}
Defines a command called **step1** that advances all threads in process 1.

capture

Assigns a command's output to a variable

Format:

capture *command*

Arguments:

command — The CLI command (or commands) whose output is being captured. If you are specifying more than one command, you must enclose them within braces (**{ }**).

Description:

The **capture** command executes *command*, capturing all output that would normally go to the console into a string. After *command* completes, it returns the string. The **capture** command lets you obtain the printed output of any CLI command so that you can assign it to a variable or otherwise manipulate it. This command is analogous to the UNIX shell's back-tick feature; that is, `*command*`.

Examples:

set save_stat [capture st]

Saves the current process status into a Tcl variable.

set vbl [capture { foreach i { 0 1 2 3 } { p an_array($i)} }]

Saves the printed output of four array elements into a Tcl variable. Here is some sample output:

```
int2_array(1) = -8 (0xfff8)
int2_array(2) = -6 (0xfffa)
int2_array(3) = -4 (0xfffc)
int2_array(4) = -2 (0xfffe)
```

Because **capture** records all of the information sent to it by the commands within the braces that follow, you do not have to use a **dlist** command.

exec cat << [capture help commands] > cli_help.txt

Writes the help text for all TotalView commands to the **cli_help.txt** file.

dactions

Displays a list of action points

Format:

dactions [*ap-id-list*] [**–at** *source-loc*] [**–enabled** | **–disabled**]

Arguments:

ap-id-list	A list of action point identifiers. If you specify action points, the information displayed is limited to these points. If no IDs are entered, TotalView displays information about all action points in the processes in the focus set. If one ID is entered, TotalView displays information for it. If more than one ID is entered, TotalView displays information for each.
–at *source-loc*	Display the action point at *source-loc*.
–enabled	Indicates that only enabled action points are shown.
–disabled	Indicates that only disabled action points are shown.

Command alias:

ac

Description:

The **dactions** command displays information about action points in the processes in the current focus. The information is printed; it is not returned.

This command also lets you obtain the action point identifier. This identifier is needed when you delete, enable, disable, import, or export action points.

NOTE The identifier is returned when the action point is created. It is also displayed when the target stops at an action point.

You can include specific action point identifiers as arguments to the command when detailed information is required. The **–enabled** and **–disabled** options restrict output to action points in one of these states.

You cannot use the **dactions** command when you are debugging a core file or before executables are loaded.

Examples:

ac –at arrays.F#56 Displays information about the action point on line 56 of **arrays.F**. (Notice that this example uses the alias instead of the full command name.) Here is the output from this command:

```
1 action point for process 1:
    1 addr=0x100012a8 [arrays.F#56] Enabled
```

dactions 1 3 Displays information about action points 1 and 3. Here is some sample output:

```
2 action points for process 1:
    1 addr=0x100012a8 [arrays.F#56] Enabled
    3 addr=0x100012c0 [arrays.F#57] Enabled
```

dfocus p1 dactions Displays information on all action points defined within process 1.

dfocus p1 dactions –enabled
Displays information on all enabled action points within process 1.

dassign

Changes the value of a scalar variable

Format:

dassign *target value*

Arguments:

target	A variable name referring to a target program scalar value.
value	A source-language expression that evaluates to a scalar value. This expression can use the name of another variable.

Command alias:

as

Description:

The **dassign** command evaluates an expression and replaces the value of a variable with the evaluated result. The target location may be a scalar variable, a dereferenced pointer variable, or an element of an array or structure.

The default focus for **dassign** is *thread*. So, if you do not change the focus, this command acts upon the *thread of interest*. If the current focus specifies a width that is wider than **t** (thread) and is not **d** (default), **dassign** iterates over the threads in the focus set and performs the assignment in each. In addition, if you use a list with the **dfocus** command, **dassign** iterates over each list member.

The CLI interprets each symbol name in the expression according to the current context. Because the value of a source variable may not have the same value across threads and processes, the value assigned can differ in your threads and processes. If the data type of the resulting value is incompatible with that of the target location, you must cast the value into the target's type. (*Casting* is described in Chapter 7 of the TOTALVIEW USER'S GUIDE.)

Here are some things you should know about assigning characters and strings:

- If you are assigning a character to a *target*, place the character value within single quotation marks; for example, **'c'**.
- You can use the standard C language escape character sequences; for example, **\n**, **\t**, and the like. These escape sequences can also be within a character or string assignment.

- If you are assigning a string to a *target*, place the string within quotation marks. However, you must "escape" the quotation marks so they are not interpreted by Tcl; for example, **\"The quick brown fox\"**.

If *value* contains an expression, the expression is evaluated by TotalView's expression system. This system is discussed in Chapter 8 of the TOTALVIEW USER'S GUIDE.

Examples:

dassign scalar_y 102

Stores the value 102 in each occurrence of variable **scalar_y** for all processes and threads in the current set.

f { p1 p2 p3 } as scalar_y 102

Stores the value 102 in each occurrence of variable **scalar_y** contained within processes 1, 2, and 3.

dattach

Brings currently executing processes under CLI control

Format:

dattach[**-g** *gid*] [**-r** *hname*] [**-e**] *fname pid-list*

Arguments:

-g *gid*
: Sets the program control group for the process being added group *gid*. This group must already exist. (The CLI **GROUPS** variable contains a list of all groups. See **dset** on page 80 for more information.)

-r *hname*
: The host on which the process is running. The CLI will launch a TotalView Debugger Server on the host machine if one is not already running there. Consult the TOTALVIEW USER GUIDE for information on the launch command used to start this server.

  Setting a host sets it for all PIDs attached to in this command. If you do not name a host machine, the CLI uses the local host.

-e
: Tells the CLI that the next argument is a filename. You need to use this argument if the filename begins with a dash or only uses numeric characters.

fname
: The name of the executable. Setting an executable here, sets it for all PIDs being attached to in this command. If you do not include this argument, the CLI tries to determine the executable file from the target process. Some architectures do not allow this to occur.

pid-list
: A list of system-level process identifiers (such as a UNIX PID) naming the processes that TotalView will control. All PIDs must reside on the same system and they will all be placed into the same program group.

  If you need to place the processes in different groups or attach to processes on more than one system, you must use multiple **dattach** commands.

Command alias:

at

Description:

The **dattach** command tells TotalView to attach to one or more processes, making it possible to continue process execution under CLI control.

This command returns the TotalView process ID (PID) as a string value. If you specify more than one process in a command, **dattach** returns a list of PIDs instead of a single value.

All processes attached to in one **dattach** command are placed in the same program group. This allows you to place all processes in a multiprocess program executing on the same system in the same program group.

If a target program has more than one executable, you must use a separate **dattach** for each.

If the *fname* executable is not already loaded, the CLI searches for it. The search will include all directories in the **EXECUTABLE_PATH** CLI state variable.

The process identifiers specified in the *pid-list* must refer to existing processes in the run-time environment. TotalView attaches to the processes, regardless of their execution states.

Examples:

dattach mysys 10020

Loads debugging information for **mysys** and brings the process known to the run-time system by PID 10020 under CLI control.

dattach -e 123 10020

Loads file 123 and brings the process known to the run-time system by PID 10020 under CLI control.

dattach -g 4 -r Enterprise myfile 10020

Loads **myfile** that is executing on the host named **Enterprise** into group 4 and brings the process known to the run-time system by PID 10020 under CLI control. If a TotalView Debugger Server (**tvdsvr**) is not running on **Enterprise**, the CLI will start it.

dattach my_file 51172 52006

Loads debugging information for **my_file** and brings the processes corresponding to PIDs 51172 and 52006 under CLI control.

```
set new_pid [dattach -e mainprog 123]
dattach -r otherhost -g $CGROUP($new_pid) -e slaveprog 456
```

Begins by attaching to **mainprog** running on the local host. It then attaches to **slaveprog** running on otherhost and inserts them both in the same program control group.

dbreak

Defines a breakpoint

Format:

Creates a breakpoint at a source location

dbreak *source-loc* [–p | –g] [[–l *lang*] –e *expr*]

Creates a breakpoint at an address

dbreak –address *addr* [–p | –g] [[–l *lang*] –e *expr*]

Arguments:

source-loc	The breakpoint location specified as a line number or as a string containing a file name, function name, and line number, each separated by **#** characters; for example, **#file#line**. Defaults are constructed if you omit parts of this three-part specification. For more information, see "Qualifying Symbol Names" on page 24.
–address *addr*	The breakpoint location specified as an absolute address in the address space of the target program.
–p	Tells TotalView to stop the process that hit this breakpoint.
–g	Tells TotalView to stop all processes in the process's program group when the breakpoint is hit. (This is normally the default. However, you can change the default by setting the STOP_ALL state variable. See **dset** on page 80 for more information.)
–l *lang*	Sets the programming language used when you are entering expression *expr*. The languages you can enter are **c c++**, **f77**, **f90**, and **asm** (for C, C++, FORTRAN 77, Fortran 9x, and assembler). If you do not specify a language, TotalView assumes that you wrote the expression in the same language as the routine at the breakpoint. Note that not all languages are supported on all systems.
–e *expr*	When the breakpoint is hit, tells TotalView to evaluate expression *expr* in the context of the thread that hit the breakpoint. The language statements and operators you can use are described in the TOTALVIEW USER'S GUIDE.

Command alias:

b

Description:

The **dbreak** command defines a breakpoint or evaluation point that is triggered when execution arrives at the specified location. The ID of the new breakpoint is returned.

Each thread stops when it arrives at a breakpoint, where it is held until a resume command is issued. The resume commands (**dstep**, **dgo**, and the like) tell the held thread to continue executing.

Specifying a procedure name without a line number tells the CLI to set an action point at the beginning of the procedure. If you do not specify a file, the default is the file associated with the current source location.

The CLI may not be able to set a breakpoint at the line you specify. This is because the compiler may not generate the information that the CLI needs. This may happen, for example, if a line does not contain an executable statement or if the line numbers generated by the compiler span source lines.

If you try to set a breakpoint on a line at which the CLI cannot stop execution, it sets one at the nearest following line where it can halt execution.

When the CLI displays information on a breakpoint's status, it displays the location where execution will actually stop.

If the CLI encounters a *stop group* breakpoint, it suspends each process in the group as well as the processes containing the triggering threads. The CLI then shows the identifier of the triggering thread, the breakpoint location, and the action point identifier.

One possibly confusing aspect of using expressions is that their syntax differs from that of Tcl. This is because you will need to embed code written in Fortran, C, or assembler within Tcl commands. In addition, your expressions will often include TotalView intrinsic functions.

Examples:

For all examples, assume the current process set is **d2.<** when the breakpoint is defined.

dbreak 12
Suspends process 2 when it reaches line 12. However, if the **STOP_ALL** state variable is set to true, all other processes in the group are stopped. In addition, if you have set the **SHARE_ACTION_POINT** state variable to true, the breakpoint is placed in every process in the group

dbreak –address 0x1000764
Suspends process 2 when address 0x1000764 is reached.

b 12 –g
Suspends all processes in the current process group when line 12 is reached.

dbreak 57 –l f9 –e { goto $63 }
Causes the thread that struck the breakpoint to transfer to line 63. The host language for this statement is Fortran 90 or Fortran 95.

dfocus p3 b 57 –e { goto $63 }
In process 3, sets the same evaluation point as the previous example.

dcont

Continues execution of processes and waits for them

Format:

dcont [–waitany]

Arguments:

–waitany	Blocks execution until a process in the target set stops. This option is not needed if you are debugging a multiprocess program.

Command alias:

co
CO

Description:

The **dcont** command continues all processes and threads in the current focus and then waits for them to stop. The **CO** command is an alias for **dfocus g dcont**, and acts as a group-resume command.

This command is defined as follows:

```
proc dcont { args } { uplevel "dgo; dwait $args"}
```

If you add the **–waitany** option, this command blocks until some process, not the whole collection, stops.

The **dcont** command alters program state by changing the state of all processes in the affected set to *running*. Only threads that are currently in the *stopped/runnable* state can actually be changed in this way. Program state is unaffected for any threads that are already running or held at a barrier.

A **dcont** command completes when all threads in the appropriate processes have stopped executing (that is, none of them is in the *running* state).

Examples:

dcont	Resumes execution of all *stopped/runnable* threads belonging to processes in the current focus. (Threads held at barriers are not affected.) The command blocks further input until all threads in all target processes stop. When the prompt is displayed, you can enter additional commands.
dfocus p1 dcont	Resumes execution of all *stopped/runnable* threads belonging to process 1. Command input is blocked until the process stops.

dfocus {p1 p2 p3} co –waitany
: Resumes execution of all *stopped/runnable* threads belonging to processes 1, 2, and 3, blocking further command input until at least one process stops.

CO –waitany
: Resumes execution of all *stopped/runnable* threads belonging to the current group, blocking further command input until at least one process stops.

ddelete

Deletes action points

Format:

Deletes the listed action points

ddelete *action-point-list*

Deletes all action points

ddelete –a

Arguments:

action-point-list	A list of the action points being deleted.
–a	Tells TotalView to delete all action points.

Command alias:

de

Description:

The **ddelete** command permanently removes one or more action points. The argument to this command can name the action points being deleted. The **–a** option indicates that the CLI should delete all action points.

The **ddelete** command affects CLI state by removing all information about the deleted action points. It does not directly affect program state unless the specified action point is a barrier; in this case, the state of processes in a held target set is changed from *stopped/held* to *stopped/runnable*.

Examples:

ddelete 1 2 3	Deletes breakpoints 1, 2, and 3.
ddelete –a	Deletes all action points associated with processes in the target focus.
dfocus {p1 p2 p3 p4} ddelete –a	Deletes all of the breakpoints associated with processes 1 through 4. Breakpoints associated with other threads are not affected.
dfocus a de –a	Deletes all action points known to the CLI.

ddetach

Detaches from processes

Format:

ddetach

Command alias:

det

Description:

The **ddetach** command detaches the CLI from all processes in the current focus. This *undoes* the effects of attaching the CLI to a running process; that is, the CLI releases all control over the process, eliminates all debugger state information related to it (including action points), and allows the process to continue executing in the normal run-time environment.

You can detach any process controlled by the CLI; the process being detached does not have to be originally loaded with a **dattach** command.

After this command executes, you are no longer able to access program variables, source location, action point settings, or other information related to the detached process.

If a single thread serves as the target set, the CLI detaches the process containing the thread.

Examples:

ddetach	Detaches the process or processes that are in the current focus.
dfocus {p4 p5 p6} det	Detaches processes 4, 5, and 6.
dfocus g2 det	Detaches all processes in the program group associated with process 2.

ddisable

Temporarily disables action points

Format:

Disables specific action points

ddisable *action-point-list*

Disables all action points

ddisable –a

Arguments:

action-point-list	A list of action points being disabled.
–a	Tells TotalView to disable all action points.

Command alias:

di

Description:

The **ddisable** command temporarily deactivates action points; it does not, however, delete them.

The first form of this command lets you explicitly name the IDs of the action points being disabled. The second form lets you disable all action points.

The **ddisable** command affects CLI state by disabling the corresponding action points. It has no effect on program state.

Examples:

ddisable 3 7	Disables the action points whose IDs are 3 and 7.
di -a	Disables all action points in the current focus.
dfocus {p1 p2 p3 p4} ddisable –a	Disables action points associated with processes 1 through 4. Action points associated with other processes are not affected.

ddown

Moves down the call stack

Format:

ddown [*num-levels*]

Arguments:

num-levels	Number of levels to move down. The default is 1.

Command alias:

d

Description:

The **ddown** command moves the selected stack frame down one or more levels. It also prints the new frame's number and function name.

Call stack movements are all relative, so **ddown** effectively "moves down" in the call stack. (If "up" is in the direction of **main()**, then "down" is back from where you started moving through stack frames.)

Frame 0 is the most recent—that is, currently executing—frame in the call stack, frame 1 corresponds to the procedure that invoked the currently executing one, and so on. The call stack's depth is increased by one each time a procedure is entered, and decreased by one when it is exited.

The command affects each thread in the target focus. Any collection of processes and threads can be specified as the target set.

In addition, the **ddown** command modifies the current list location to be the current execution location for the new frame; this means that a **dlist** command displays the code surrounding this new location.

The context and scope changes made by this command remain in effect until the CLI executes a command that modifies the current execution location (for example, **dstep**), or until you enter a **dup** or **ddown** command.

If you tell the CLI to move down more levels than exist, the CLI simply moves down to the lowest level in the stack (which was the place where you began moving through the stack frames).

Examples:

ddown

Moves down one level in the call stack. As a result, for example, **dlist** commands that follow will refer to the procedure that invoked this one. Here is an example of what is printed after you enter this command:

```
0 check_fortran_arrays_ PC=0x10001254,
    FP=0x7fff2ed0 [arrays.F#48]
```

d 5

Moves the current frame down five levels in the call stack.

denable

Reenables action points

Format:

Reenables specific action points

denable *action-point-list*

Reenables all disabled action points in the current focus

denable –a

Arguments:

action-point-list	The identifiers of the action points being enabled.
–a	Tells TotalView to enable all action points.

Command alias:

en

Description:

The **denable** command reactivates action points that were previously disabled. The **–a** option tells the CLI to enable all action points in the current focus.

The easiest way of knowing what these IDs are is to retain the value returned by the **dbreak** command or extract them from the **dactions** command.

Examples:

denable 3 4	Enables two previously identified action points. These action points were previously disabled with the **ddisable** command.
dfocus {p1 p2} denable –a	Enables all action points associated with processes 1 and 2. Settings associated with other processes are not affected.
en –a	Enables all action points associated with the current focus.
f a en –a	Enables all actions points in all processes.

dfocus

Changes the current Process/Thread set

Format:

Changes the target of CLI commands to this P/T set

dfocus *p/t-set*

Executes a command within this P/T set

dfocus *p/t-set command*

Arguments:

p/t-set	A set of processes, threads, and arenas. This set defines the target upon which CLI commands will act.
command	A CLI command, which when it executes, operates upon its own local focus.

Command alias:

f

Description:

The **dfocus** command changes the set of processes, threads, and groups upon which a command will act. This command can change the focus for all commands that follow or just the command that immediately follows.

The **dfocus** command always expects a P/T value as its first argument. This value can either be a single arena specifier or a list of arena specifiers. The default focus is **d1.<**, which selects the first user thread. The **d** (for default) indicates that each CLI command is free to use its own default width.

If you enter an optional *command*, the focus is set temporarily, and *command* is executed in the new focus. After *command* executes, the focus is restored to its original value. *command* may be a single command or a list.

If you use a *command* argument, **dfocus** returns the result of the command. If you do not enter a command, **dfocus** returns the focus as a string value.

The **dfocus** command affects CLI state by replacing the processes and threads previously used as the current set, with those specified in the argument. It does not affect program state in any way.

Examples:

dfocus g dgo — Continues the TotalView group containing the focus process.

dfocus p3 { dhalt ; dwhere } — Stops process 3 and displays backtraces for each of its threads.

dfocus 2.3 — Sets the focus to thread 3 of process 2, where the "2" and the "3" are TotalView's process and thread identifier values. The focus is set to **d2.3**.

dfocus 3.2
dfocus .5 — Sets, then resets command focus. A focus command that includes a dot and omits the process value tells the CLI to use the current process. Thus, this sequence of commands changes the focus to *process* 3, *thread* 5 (**d3.5**).

dfocus g dstep — Steps the current group. Note that while the thread of interest is determined by the current focus, the command acts upon the entire group containing that thread.

dfocus {p2 p3} {dwhere ; dgo} — Performs a backtrace on all threads in processes 2 and 3 and then continues the processes.

f 2.3 { f p w ; f t s ; g } — Executes a backtrace on all the threads in process 2, steps thread 3 in process 2 (without running any other threads in the process), and continues the process.

dfocus p1 — Changes the current focus to include just those threads currently in process 1.

dfocus a — Changes the current set to include all threads in all processes. When you execute this command, you will notice that your prompt changes to **a1.<**. This command alters the CLI's behavior so that actions that previously operated on a thread now apply to all threads in all processes.

dgo

Resumes execution of target processes

Format:

dgo

Command alias:

g
G

Description:

The **dgo** command continues all processes and threads in the current focus. If a target process does not exist, this command creates it, passing it the default command arguments. The **G** command is an alias for **dfocus g dgo**, and acts as a group-resume command.

This command has no arguments.

The **dgo** command alters program state by changing the state of all processes in the affected set to *running*. Only threads that are currently in the *stopped/runnable* state can actually be changed in this way, however. Program state is unaffected for any threads that are already running, held at a barrier, and so on.

A **dgo** command cannot be applied when you are debugging a core file, nor can you use it before the CLI loads an executable and starts executing it.

Examples:

dgo	Resumes execution of all *stopped/runnable* threads belonging to processes in the current focus. (Threads held at barriers are not affected.)
G	Resumes execution of all threads in the current process group.

dhalt

Suspends execution of target processes

Format:

dhalt

Command alias:

h
H

Description:

The **dhalt** command stops all processes and threads in the current focus. The **H** command is an alias for **dfocus g dhalt**, and acts as a group-stop command.

The command has no arguments.

The **dhalt** command alters program state by changing the state of all processes in the affected set to *stopped/runnable*. Only threads that are currently in the *running* state can actually be halted, however. Program state is not altered for any threads that are already stopped, held at a barrier, and so on.

Examples:

dhalt	Suspends execution of all *running* threads belonging to processes in the current focus. (Threads that are held at barriers are not affected.)
f 1.1 h	Suspends execution of thread 1 in process 1. Note the difference between this command and **f 1.< dhalt**. If the focus is set as thread-level, this command will halt the first user thread, which is probably thread 1.

dkill

Terminates execution of target processes

Format:

dkill

Command alias:

k

Description:

The **dkill** command terminates all target processes in the current focus.

This command has no arguments.

Because the executables associated with the defined processes are still "loaded," typing the **drun** command restarts the processes.

The **dkill** command alters program state by terminating all processes in the affected set. In addition, TotalView destroys any spawned processes when the process that created them is killed. Only the initial process can be restarted by using the **drun** command.

Examples:

dkill	Terminates all threads belonging to processes in the current focus.
dfocus {p1 p3} dkill	Terminates all threads belonging to processes 1 and 3.

dlist

Displays source code lines

Format:

Displays code relative to the current list location

dlist [–n *num-lines*]

Displays code relative to a named place

dlist *source-loc* [–n *num-lines*]

Displays code relative to the current execution location

dlist –e [–n *num-lines*]

Arguments:

–n *num-lines*	Requests that this number of lines be displayed rather than the default value. (The default is the value of the **MAX_LIST** variable.) If *num-lines* is negative, lines before the source display location are shown, and additional **dlist** commands will show preceding lines in the file rather than succeeding lines. For more information, see "dset" on page 80. This option also sets the value of the **MAX_LIST** variable to *num-lines*.
source-loc	Sets the location at which the CLI begins displaying information. This location is specified as a line number or as a string containing a file name, function name, and line number, each separated by **#** characters. For example: **#file#func#line**. For more information, see "Qualifying Symbol Names" on page 24. Defaults are constructed if you omit parts of this specification.
–e	Sets the source display location to include the current execution point of the thread of interest. If you used **dup** and **ddown** commands to select a buried stack frame, this location includes the PC (program counter) for that stack frame.

Command alias:

l

Description:

The **dlist** command displays lines relative to a place in the source code. (This place is called the *source display location*.) This information is printed; it

is not returned. If neither *source-loc* nor **–e** is specified, the command continues where the previous list command left off. If a new file or function name is specified (not just a new line number), the source display location setting is remembered; setting the focus to a different thread will not change it. To display the thread's execution point, use **dlist –e**.

When a file or procedure is named, the listing begins at the file or procedure's first line.

The first time you use the **dlist** command after you focus on a different thread—or after the focus thread runs and stops again—the source display location changes to include the current execution point of the new focus thread.

Tabs in the source file are expanded as blanks in the output. The tab stop width is controlled by the **TAB_WIDTH** variable, which has a default value of 8. If **TAB_WIDTH** is set to -1, no tab processing is done, and tabs are displayed using their ASCII value.

All lines are shown with a line number and the source text for the line. The following symbols are also used:

@	An action point is set at this line.
>	The PC for the current stack frame is at the indicated line and this is the leaf frame.
=	The PC for the current stack frame is at the indicated line and this is a buried frame.

These correspond to the marks shown in the backtrace displayed by **dwhere** that indicates the selected frame.

Here are some general rules:

- The initial *current list location* is **main()**.
- The current list location is *per thread*.
- The current list location is set to the current execution location when a thread stops.

If the *source-loc* argument is not fully qualified, the CLI looks for it in the directories named in the CLI **EXECUTABLE_PATH** state variable.

When you do use a procedure name without also using the **–n** option, the CLI displays the entire procedure.

This command has no effect on program state.

Examples:

These examples assume that **MAX_LIST** is at its initial value of 20.

dlist — Displays 20 lines of source code, beginning at the current list location. The list location is incremented by 20 when the command completes.

dlist 10 — If this command is executed after the previous example, displays 10 lines, starting with line 20 of the file corresponding to the current list location (that is, 10 lines before the current location). The list location is changed to line 30.

dlist –n 10 — Displays 10 lines, starting with the current list location. The value of the list location is incremented by 10.

dlist –n -50 — Displays source code preceding the current list location; 50 lines are shown, ending with the current source code location. The list location is decremented by 50.

dlist #do_it — Displays all source code lines for procedure **do_it**. The list location is changed so that it is the first line following the end of the procedure.

dfocus 2.< dlist #do_it — Displays all source code lines for the routine **do_it** associated with process 2. If the current source file were named **foo**, this could also be specified as **dlist #foo#do_it**, naming the executable for process 2.

dlist –e — Displays 20 lines starting 10 lines above the current execution location. If more than one location is current for the target set, the CLI chooses the one it thinks you will be most interested in. The list location is updated to that line plus 20.

f 1.2 l –e — Starts the display at the current execution location of thread 2 in process 1.

dfocus 1.2 dlist –e –n 10

Produces essentially the same listing as the previous example, differing in that 10 lines are displayed.

dlist #do_it.f#80 –n 10

Displays 10 lines, starting with line 80 in file **do_it.f**. The list location is updated to line 90.

dload

Loads debugging information

Format:

dload [-g *gid*] [-r *hname*] [-e] *executable*

Arguments:

-g *gid*	Sets the program control group for the process being added to group *gid*. This group must already exist. (The CLI **GROUPS** variable contains a list of all groups.)
-r *hname*	The host on which the process is running. The CLI will launch a TotalView Debugger Server on the host machine if one is not already running there. Consult the TOTALVIEW USER GUIDE for information on the launch command used to start this server.
-e	Tells the CLI that the next argument is a filename. You need to use this argument if the filename begins with a dash or only uses numeric characters.
executable	A fully or partially qualified file name for file corresponding to the target program.

Command alias:

lo

Description:

The **dload** command creates a new TotalView process object for *executable*. You will actually create the process by using the **drun** on page 78 or **drerun** on page 77 commands. (The descriptions for these commands explain how the CLI passes arguments to a process when it is launched.) The **dload** command returns the TotalView ID for the new object.

Examples:

dload do_this	Loads the debugging information for executable **do_this** into the CLI. After this command completes, the target process does not yet exist and no address space or memory is allocated to it.
lo -g 3 -r other_computer do_this	Loads the debugging information for executable **do_this** on the **other_computer** machine into the CLI. This process is placed into group 3.

f g3 lo -r other_computer do_this

Does not do what you would expect it to do because the **dload** command ignores the focus command. You should note that this group 3 refers to the debugger ID while the 3 in the previous example refers to the group whose GID is 3.

dnext

Steps source lines, stepping over subroutines

Format:

dnext [*num-steps*]

Arguments:

num-steps	An integer number greater than 0, indicating the number of source lines to be executed.

Command alias:

n
N

Description:

The **dnext** command executes source lines; that is, it advances the program by steps (source line statements). However, if a statement in a source line invokes a subfunction, **dnext** executes the subfunction as if it were one statement; that is, it steps *over* the call.

The **N** command is an alias for **dfocus g dnext**, and acts as a group-step command.

The optional *num-steps* argument tells the CLI how many **dnext** operations it should perform. If you do not specify *num-steps*, the default is 1.

The **dnext** command iterates over the arenas in its focus set, performing a thread-level, process-level, or group-level step in each arena, depending on the width of the arena. The default width is process (**p**).

For more information on stepping in processes and threads, see "dstep" on page 86.

Examples:

dnext	Steps one source line.
n 10	Steps ten source line.
N	Steps the entire group one source line.
f t n	Steps the entire thread.
dfocus 3. dnext	Steps process 3 one step.

dnexti

Steps machine instructions, stepping over subroutines

Format:

dnexti [*num-steps*]

Arguments:

num-steps	An integer number greater than 0, indicating the number of instructions to be executed.

Command alias:

ni
NI

Description:

The **dnexti** command executes machine-level instructions; that is, it advances the program by steps (instructions). However, if the instruction invokes a subfunction, **dnexti** executes the subfunction as if it were one instruction; that is, it steps *over* the call.

The **NI** command is an alias for **dfocus g dnexti**, and acts as a group-step command.

The optional *num-steps* argument tells the CLI how many **dnexti** operations it should perform. If you do not specify *num-steps*, the default is 1.

The **dnexti** command iterates over the arenas in the focus set, performing a thread-level, process-level, or group-level steps in each arena, depending on the width of the arena. The default width is process (**p**).

For more information on stepping in processes and threads, see **dstep on page 86**.

Examples:

dnexti	Steps one machine-level instruction.
ni 10	Steps ten machine-level instructions.
NI	Steps the entire group one machine-level instruction.
f t n	Steps the entire thread one machine-level instruction.
dfocus 3. dnexti	Steps process 3 one machine-level instruction.

dprint

Evaluates and displays information

Format:

Prints the value of a variable

dprint *variable*

Prints the value of an expression

dprint *expression*

Arguments:

variable	A variable whose value will be displayed. The *variable* argument can be local to the current stack frame or a global.
expression	A source-language expression to be evaluated and printed. Because *expression* must also conform to Tcl syntax, you must place it within quotes if it includes any blanks, and it must be enclosed in braces ({}) if it includes square brackets ([ ]), dollar signs ($), quote characters ("), or any other Tcl special characters. *expression* cannot contain calls to assembler, Fortran, C, or C++ functions.

Command alias:

p

Description:

The **dprint** command evaluates and displays a variable or an expression. The CLI interprets the expression by looking up the values associated with each symbol and applying the operators. The result of an expression can be a scalar value or an aggregate (array, array slice, or structure).

As the CLI displays data, it passes the data through a simple *more* process that prompts you after each screen of text is displayed. After a screen of data is displayed, you can press the **Enter** key to tell the CLI to continue displaying information. Typing **q** tells the CLI to stop printing this information.

The **dprint** command has no effect on program state or CLI state.

Since a considerable amount of output can be generated, you may want to use the **capture** command described on page 42 to save the output into a variable.

Structure output appears with one field printed per line. For example:

```
sbfo = {
  f3 = 0x03 (3)
  f4 = 0x04 (4)
  f5 = 0x05 (5)
  f20 = 0x000014 (20)
  f32 = 0x00000020 (32)
 }
```

Arrays are printed in a similar manner. For example:

```
foo = {
  [0][0] = 0x00000000 (0)
  [0][1] = 0x00000004 (4)
  [1][0] = 0x00000001 (1)
  [1][1] = 0x00000005 (5)
  [2][0] = 0x00000002 (2)
  [2][1] = 0x00000006 (6)
  [3][0] = 0x00000003 (3)
  [3][1] = 0x00000007 (7)
 }
```

Each expression is evaluated in the context of each thread in the target focus. Thus, the overall format of **dprint** output is as follows:

```
first process/thread group:
    expression result

second process/thread group:
    expression result

...

last process/thread group:
    expression result
```

Examples:

dprint scalar_y	Displays the values of variable **scalar_y** within all processes and threads in the current set.
p argc	Displays the value of **argc**.
p argv	Displays the value of **argv**, along with the first string to which it points.

p { argv[argc-1]} Prints the value of **argv[argc-1]**. If the execution point is in **main()**, this is the last argument passed to **main()**.

dfocus p1 dprint scalar_y
Displays the values of variable **scalar_y** for the threads in process 1.

f 1.2 p arrayx Displays the values of the array **arrayx** for just the second thread in process 1.

for {set i 0} {$i < 100} {incr i} {p argv\[$i\]}
If **main()** is in the current scope, prints the program's arguments followed by the program's environment strings.

drerun

Restarts processes

Format:

drerun [*args*]

Arguments:

args	The arguments to be used for restarting a process.

Command alias:

rr

Description:

The **drerun** command restarts the process that is in the current focus set from its beginning. **drerun** uses the arguments stored in the **ARGS** and **ARGS_DEFAULT** state variables. These are set every time the process is run with different arguments. Consequently, if you do not specify the arguments to be used when restarting the process, the CLI uses the arguments specified when the process was previously run. (See **drun** on page 78 for more information.)

The **dererun** command differs from the **drun** command in that

- If you do not specify an argument, **drerun** uses the default values. In contrast, the **drun** command clears the argument list for the program. This means that you cannot use an empty argument list with the **drerun** command to tell the CLI to restart a process and expect that no arguments will be used.
- If the process already exists, **drun** will not restart it. (If you must the **drun** command, you must first kill the process.) In contrast, the **drerun** command will kill and then restart the process.

Examples:

drerun	Reruns the current process. Because arguments are not used, the process is restarted using its previous values.
rr –firstArg an_argument –aSecondArg a_second_argument	Reruns the current process. The default arguments are not used because replacement arguments are specified.

drun

Starts or restarts processes

Format:

drun [*cmd_arguments*] [< *infile*] [> *outfile*]

Arguments:

cmd_arguments	The argument list passed to the process.
infile	If specified, indicates a file from which the CLI will read information.
outfile	If specified, indicates the file into which the CLI will write information.

Command alias:

r

Description:

The **drun** command launches each process in the current focus and starts it running. The command arguments are passed to the processes, and I/O redirection for the target program, if specified, will occur. Later in the session, you can use the **drerun** command to restart the program.

In addition, the CLI uses the following state variables to hold the "default" argument list for the processes.

ARGS_DEFAULT	The CLI sets this variable if you use the **–a** option when you started the CLI or TotalView. (This option passes command lines arguments that TotalView will use when it invokes a process.) It holds the default arguments that are passed to a process when the process has no default arguments of its own.
ARGS(*n*)	The command-line arguments are stored in this array variable. The index to locations within the array is the process ID *n*. This variable has a process's default arguments. It is always set by the **drun** command, and it is also set by the **drerun** command when it is used with arguments.

If more than one process is launched with a single **drun** command, each receives the same command-line arguments.

In addition to setting these variables by using the **–a** option or specifying *cmd_arguments* when you use this or the **drerun** command, you can modify these variables directly with the **dset** and **dunset** commands.

A reissued **drun** command can only be applied to initial processes, not to processes that were spawned by the target program. Further, each initial process must be terminated; if a process is not terminated, you are told to kill it and retry. (You can, of course, use the **rerun** command.)

The first time the **drun** command is issued, arguments are copied to program variables, and any I/O redirection is initiated. When the command is reissued for processes that were started previously—or issued for the first time for a process that was attached using the **dattach** command—the CLI reinitializes your program.

Debugging information includes information about the symbol table, dynamic linking, compiler optimizations, and so on. TotalView uses this information to initialize CLI state, machine state, and (indirectly) program state.

Examples:

drun	Tells the CLI to begin executing all threads belonging to processes represented in the current focus.
f {p2 p3} drun	Begins execution of all threads in processes 2 and 3.
f 4.2 r	Begins execution of all threads belonging to process 4. Note that this might be the same as **f p4 drun** if the current focus is set to default (**d**) or process (**p**) width.
dfocus a drun	Restarts execution of all threads known to the CLI. If they were not previously killed, you are told to use the **dkill** command and then try again.
drun < in.txt	Restarts execution of all threads in the current focus, setting them up to get standard input from file **in.txt**.

dset

Changes or views CLI state variables

Format:

Changes a CLI state variable

dset *debugger-var value*

Views current CLI state variables

dset [*debugger-var*]

Arguments:

debugger-var	Name of a CLI state variable.
value	Value to be assigned to *debugger-var.*

Description:

The **dset** command sets the value of CLI debugger variables.

When no arguments are specified, **dset** displays the names and current values for all TotalView CLI state variables. If you use only one argument, the CLI returns and displays the variable's value.

The second argument defines the value that will replace a variable's previous value. It must be enclosed in quotes if it contains more than one word.

The state variables are

ARGS(*n*)	An array that contains the arguments used when a process is started.
ARGS_DEFAULT	Contains the default arguments used when a process is started.
BARRIER_STOP_ALL	Contains the default value for the **STOP_ALL** variable on newly created barrier points. If this variable is set to true, it indicates the CLI will stop all group members when one process hits the barrier. If it is false, the CLI lets the other processes continue to run.
CGROUP(*dpid*)	Contains the program control group for the process with the TotalView ID **dpid**. Setting this variable moves process **dpid** into a different control group. For example, the following command moves process 3 into the same group as process 1:

```
dset CGROUP(3) $CGROUP(1)
```

EXECUTABLE_PATH
: Has a list containing the directories that the CLI searches when it looks for source and executable files.

GROUP(*gid*)
: Contains a list containing the TotalView IDs for all members in group **gid**. The first element indicates what kind of group it is and can be:

control	Program control group
share	Breakpoint share group

GROUPS
: Contains a list containing the IDs for all groups in TotalView.

LINES_PER_SCREEN
: Defines the number of lines shown before the CLI stops printing information and displays its *more* prompt. The following values have special meaning:

0	No *more* processing occurs, and the printing does not stop when the screen fills with data.
NONE	This is a synonym for 0.
AUTO	The CLI uses the **tty** settings to determine the number of lines to display. This may not work in all cases. For example Emacs sets the **tty** value to 0. If **AUTO** works improperly, you will need to explicitly set a value.

MAX_LIST
: Defines the number of lines displayed in response to a **dlist** command.

PROMPT
: Defines the CLI prompt. If you are including a variable or a value returned from a function in the prompt, be sure to enclose it within brackets ([ ]) so that the Tcl interpreter knows to substitute the returned value.

PTSET
: Defines the current focus.

SHARE_ACTION_POINT
: Contains the default value for TotalView's internal **share_in_group** flag for newly created action points. If this value is true, an action point will be active across the group. If it is false, an action point will be only active in the process upon which it is set.

This variable affects the default width of the arenas the CLI uses in setting action points. For example, if the arena width is **d**, it controls whether the CLI will set a group or process action point.

STOP_ALL — Controls the default value for the **–p** and **–g** options to the **dbreak** and **dwatch** commands. If this variable is set to true, the CLI assumes that you are setting a group action point when you omit these options. If you set it to false, the CLI assumes that you are setting a process action point.

TAB_WIDTH — Indicates the number of spaces used to simulate a tab character when the CLI displays information.

TOTALVIEW_ROOT_PATH
Names the directory in which the TotalView executable is located.

TOTALVIEW_TCLLIB_PATH
Contains a list containing the directories that the CLI searches for TCL library components.

TOTALVIEW_VERSION
Contains the version number and the type of computer architecture upon which TotalView is executing.

VERBOSE — Controls the error message information displayed by the CLI. The values that this variable can take are as follows:

INFO	Prints error, warnings, and informational messages. Informational message include data on dynamic libraries and symbols.
WARNING	Only print errors and warnings.
ERROR	Only print error messages.
SILENT	Do not print error, warning, and informational messages. This also shuts off the printing of results from CLI commands. Because no information is printed, this should only be used when running the CLI in batch mode.

The following table lists the default and permitted values for all CLI variables:

TABLE 2: Defaults and Permitted Values for CLI Variables

Debugger Variable	Permitted Values	Default
ARGS	A string	none
ARGS_DEFAULT	A number	none
BARRIER_STOP_ALL	True or false	true
CGROUP	A number	none
EXECUTABLE_PATH	Any valid directory or directory path; to include the current setting, use $EXECUTABLE_PATH	./:$PATH
GROUP	A Tcl list; each list element has two components: the group type and the group ID This is a readonly value and cannot be set	none
GROUPS	A Tcl list of IDs; this is a read-only value and cannot be set	none
LINES_PER_SCREEN	A positive integer, or the AUTO or NONE values	AUTO
MAX_LIST	A positive integer	20
PROMPT	Any string. If you wish to access the value of PTSET, you must place the variable within braces; that is, [dset PTSET]	{[dfocus]> }
PTSET	This is a read-only value and cannot be set	d1.<
SHARE_ACTION_POINT	True or false	true
STOP_ALL	True or false	true

TABLE 2: Defaults and Permitted Values for CLI Variables (cont.)

Debugger Variable	Permitted Values	Default
TAB_WIDTH	A positive number; a value of -1 indicates no tab expansion	8
TOTALVIEW_ROOT_PATH	The location of the TotalView installation directory; this is a read-only variable and cannot be set	
TOTALVIEW_TCLLIB_PATH	Any valid directory or directory path; to include the current setting, use $TOTALVIEW_TCLLIB_PATH	The directory containing the CLI Tcl libraries
TOTALVIEW_VERSION	This is a read-only value and cannot be set	
VERBOSE	INFO, WARNING, ERROR, and SILENT	INFO

Examples:

dset PROMPT "Fixme% "
Sets the prompt to be **Fixme%** followed by a space.

dset
Displays all CLI state variables and their current settings.

dset VERBOSE
Displays the current setting for output verbosity.

dset EXECUTABLE_PATH ../test_dir;$EXECUTABLE_PATH
Places **../test_dir** at the beginning of the previous value for the executable path.

dstatus

Shows current status of processes and threads

Format:

dstatus

Command alias:

st
ST

Description:

The **dstatus** command prints information about the current state of each process and thread in the current focus. The **ST** command is an alias for **dfocus g dstatus**, and acts as a group-status command.

If you have not changed the focus, the default width is "process". In this case, **dstatus** shows the status for each thread in process 1. In contrast, if you set the focus to **g1.<**, the CLI displays the status for every thread in the group containing process 1.

The command completes as soon as the state data is accessed and reported to the user.

Examples:

dstatus — Displays the status of all threads in the current focus. For example:

```
1.1:   16641.1   Stopped, PC=0x0fa56648
```

f a st — Displays the status for all threads in all processes.

f p1 st — Displays the status of the threads associated with process 1. If the focus is at its default (**d1.<**), this is the same as typing **st**.

ST — Displays the status of all threads in the group containing the focus process. For example:

```
Threads in process 1:
1.1: 41.1 Stopped, PC=0x0fa56648
Threads in process 3:
3.1: 76.1 Stopped, PC=0x10001174, [mpi.c#37]
Threads in process 4:
4.1: 60.1 Stopped, PC=0x10001174, [mpi.c#37]
Threads in process 5:
5.1: 50.1 Stopped, PC=0x10001174, [mpi.c#37]
Threads in process 6:
6.1: 56.1 Stopped, PC=0x10001174, [mpi.c#37]
```

dstep

Steps lines, stepping into subfunctions

Format:

dstep [*num-steps*]

Arguments:

num-steps	An integer number greater than 0, indicating the number of source lines to be executed.

Command alias:

s
S

Description:

The **dstep** command executes source lines; that is, it advances the program by steps (source lines). If a statement in a source line invokes a subfunction, **dstep** steps into the function.

The **S** command is an alias for **dfocus g dstep**, and acts as a group-step command.

The optional *num-steps* argument tells the CLI how many **dstep** operations it should perform. If you do not specify *num-steps*, the default is 1.

The **dstep** command iterates over the arenas in the focus set by doing a thread-level, process-level, or group-level step in each arena, depending on the width of the arena. The default width is process (**p**).

If the width is process, the **dstep** command affects the entire process containing the thread to be stepped. Thus, while only one thread is stepped forward, all other threads contained in the same process will also resume. In contrast, the **dfocus t dstep** command tells the CLI that it should just step the *thread of interest*.

NOTE On systems having an identifiable manager thread, the "dfocus t dstep" command allows the manager thread to run as well as the thread of interest.

If a secondary thread hits a temporary breakpoint, the CLI steps over it and operation is resumed until the thread of interest gets there.

Stepping behavior differs in threads, processes, and groups, as follows:

thread	Only the thread of interest is allowed to run. (This is not supported on all systems.)

process The entire process is allowed to run, and execution continues until the thread of interest arrives at the next statement. A temporary breakpoint is planted at this location for the duration of the command. If another thread reaches the goal breakpoint first, that thread steps over it and the process is continued again, until the thread of interest reaches the goal.

group The group is examined and each process that has a thread stopped at the same location as the thread of interest is identified (that is, it is a "matching" process). One "matching" thread from each matching process is selected. TotalView then runs all processes in the group, and waits until the thread of interest arrives at its goal location and also until each selected thread arrives there. If some thread hits a different breakpoint, the step ends.

For additional information on *stepping*, see Chapter 6 of the TOTALVIEW USER'S GUIDE.

The **dstep** command alters program state by changing the state of the target thread to *running*, after establishing an implicit breakpoint after the appropriate number of source lines. The state of all other threads in the affected set is changed to *running* as well. Since no implicit breakpoint is established for these threads, their state is subject to the normal changes caused by execution. They may advance many source lines before their execution is halted.

Examples:

dstep Executes the next source line, stepping into any procedure call that is encountered. Only the current thread is stepped.

s 15 Executes the next 15 source lines.

f p1.2 dstep Steps thread 2 in process 1 by one source line. This also resumes execution of all other threads in process 1; they are halted as soon as thread 2 in process 1 executes its statement.

f t1.2 s Steps thread 2 in process 1 by one source line. No other threads in process 1 execute.

dstepi

Steps machine instructions, stepping into subfunctions

Format:

dstepi [*num-steps*]

Arguments:

num-steps	An integer number greater than 0, indicating the number of instructions to be executed.

Command alias:

si
SI

Description:

The **dstepi** command executes assembler instruction lines; that is, it advances the program by steps (instructions). If the instruction invokes a subfunction, **dstepi** steps into the function. The **SI** command is an alias for **dfocus g dstepi**, and acts as a group-step command.

The optional *num-steps* argument tells the CLI how many **dstepi** operations it should perform. If you do not specify *num-steps*, the default is 1.

For more information, see **dstep on page 86**.

Examples:

dstepi	Executes the next machine instruction, stepping into any procedure call that is encountered. Only the current thread is stepped.
si 15	Executes the next 15 instructions.
f p1.2 dstepi	Steps thread 2 in process 1 by one instruction. This also resumes execution of all other threads in process 1; they are halted as soon as thread 2 in process 1 executes its instruction.
f t1.2 si	Steps thread 2 in process 1 by one instruction. No other threads in process 1 execute.

dunset

Restores default settings for state variables

Format:

Restores a CLI variable to its default value

dunset *debugger-var*

Restores all CLI variables to their default values

dunset –all

Arguments:

debugger-var	Name of the CLI state variable whose default setting is being restored.
–all	Restores the default settings of all CLI state variables.

Description:

The **dunset** command reverses the effects of any previous **dstep** commands, restoring CLI state variables to their default settings.

NOTE **Because user-defined variables have no default values, they are deleted.**

Tcl variables (those created using the Tcl **set** command) are, of course, unaffected by this command.

If you use the **–all** option, the **dunset** command affects all changed CLI state variables, restoring them to the settings that existed when the CLI session began. Similarly, specifying *debugger-var* tells the CLI to restore that one variable.

Examples:

dunset PROMPT	Restores the prompt string to its default setting; that is, **{[dfocus]> }**.
dunset –all	Restores all CLI state variables to their default settings.

dup

Moves up the call stack

Format:

dup [*num-levels*]

Arguments:

num-levels	Number of levels to move up. The default is 1.

Command alias:

u

Description:

The **dup** command moves the current stack frame up one or more levels. It also prints the new frame number and function.

Call stack movements are all relative, so **dup** effectively "moves up" in the call stack. ("Up" is in the direction of **main()**.)

Frame 0 is the most recent—that is, currently executing—frame in the call stack, frame 1 corresponds to the procedure that invoked the currently executing one, and so on. The call stack's depth is increased by one each time a procedure is entered, and decreased by one when it is exited. The effect of **dup** is to change the dynamic context—and hence the static scope of symbols—of commands that follow. For example, moving up one level lets you access variables that are local to the procedure that called the current one.

Each **dup** command updates the frame location by adding the appropriate number of levels.

The **dup** command also modifies the current list location to be the current execution location for the new frame, so a subsequent **dlist** displays the code surrounding this location. Entering **dup 2** (while in frame 0) followed by a **dlist**, for instance, displays source lines centered around the location from which the current routine's parent was invoked. These lines will be in frame 2.

Examples:

dup

Moves up one level in the call stack. As a result, subsequent **dlist** commands refer to the procedure that invoked this one. After this command executes, it displays information about the new frame. For example:

```
1 check_fortran_arrays_ PC=0x10001254,
    FP=0x7fff2ed0 [arrays.F#48]
```

dfocus p1 u 5

Moves up five levels in the call stack for each thread involved in process 1. If fewer than five levels exist, the CLI moves up as far as it can.

dwait

Blocks command input until target processes stop

Format:

dwait

Description:

The **dwait** command tells the CLI to wait for all threads in the current focus to stop or exit. Generally, this command treats the focus identically to other CLI commands. However, because TotalView uses a synchronous stop model, the affect of **dfocus t dwait** and **dfocus p dwait** are indistinguishable.

If you interrupt this command—typically by entering Ctrl-C—the CLI manually stops all processes in the current focus before it returns.

Unlike most other CLI commands, this command blocks additional CLI input until the blocking action is complete.

Examples:

dwait — Blocks further command input until all processes in the current focus have stopped (that is, none of their threads are still **running**).

dfocus p1 p2 dwait — Blocks command input until processes 1 and 2 stop.

dwatch

Defines a watchpoint

Format:

Defines a watchpoint for a variable

dwatch *variable* [**–length** *byte-count*] [**–p** | **–g**]
[[**–l** *lang*] **–e** *expr*] [**–t** *type*]

Defines a watchpoint for an address

dwatch –address *addr* **–length** *byte-count* [**–p** | **–g**]
[[**–l** *lang*] **–e** *expr*] [**–t** *type*]

Arguments:

variable	A symbol name corresponding to a scalar or aggregate identifier, an element of an aggregate, or a dereferenced pointer.
–address *addr*	An absolute address in the file.
–length *byte-count*	The number of bytes to watch. If a variable is named, the number of bytes watched is the variable's byte length. If you are watching a variable, you only need to specify the amount of storage to watch if you want to override the default value. (The default value is the variable's defined size.)
–p	Tells TotalView to stop the process that hit this watchpoint.
–g	Tells TotalView to stop all processes in the process's program group when the watchpoint is hit. (This is the default.)
–l *lang*	Specifies the language used when writing an expression. The values you can use for *lang* are **c**, **c++**, **f7**, **f9**, and **asm**, for C, C++, FORTRAN 77, Fortran-9x, and assembler, respectively. If you do not use a language code, TotalView picks one based on the variable's type. If only an address is used, TotalView uses the C language. Not all languages are supported on all systems.
–e *expr*	When the watchpoint is triggered, evaluates *expr* in the context of the thread that hit the watchpoint. In

most cases, you need to enclose the expression within braces ({ }).

–t *type* The data type of **$oldval**/**$newval** in the expression.

Command alias:

wa

Description:

A **dwatch** command defines a watchpoint on a memory location where the specified variables are stored. The watchpoint triggers whenever the value of the variables changes.

NOTE Watchpoints are not available on Alpha Linux.

The watched variable can be a scalar, array, record, or structure object, or a reference to a particular element in an array, record, or structure. It can also be a dereferenced pointer variable.

The CLI lets you obtain a variable's address in the following two ways if your application demands that you specify a watchpoint with an address instead of a variable name:

- **dprint** *&variable*
- **dwhat** *variable*

The **dprint** command displays an error message if the variable is within a register.

NOTE In some cases, you will not be able to obtain an address from within the CLI. If this occurs, you need to obtain it by using TotalView's Graphic Interface. Chapter 8 of the TOTALVIEW USER'S GUIDE contains much additional information on watchpoints.

If you do not use the **–length** modifier, the CLI uses the length attribute from the program's symbol table. This means that the watchpoint applies to the data object named; that is, specifying the name of an array lets you watch all elements of the array. Alternatively, you can specify that a certain number of bytes be watched, starting at the named location.

NOTE In all cases, the CLI watches addresses. If you specify a variable as the target of a watchpoint, the CLI resolves the variable to an absolute address. If you are watching a local stack variable, the position being watched is just where the variable happened to be when space for the variable was allocated.

The target focus establishes the processes (not individual threads) for which the watchpoint is in effect.

The CLI prints a message showing the action point identifier, the location being watched, the current execution location of the triggering thread, and the identifier of the triggering threads.

One possibly confusing aspect of using expressions is that their syntax differs from that of Tcl. This is because you will need to embed code written in Fortran, C, or assembler within Tcl commands. In addition, your expressions will often include TotalView intrinsic functions.

Examples:

For these examples, assume that the current process set at the time of the **dwatch** command consists only of process 2, and that **p** is a global variable that is a pointer.

dwatch *p
Watches the address stored in the pointer **p** at the time the watchpoint is defined, for changes made by process 2. Only process 2 is stopped. Note that the watchpoint location does not change when the value of **p** changes.

dwatch { * p }
Performs the same action as the previous example. Because the argument to **dwatch** contains a space, Tcl requires that you place the argument within braces.

dfocus {p2 p3} wa *p
Watches the address pointed to by **p** in processes 2 and 3. Because this example does not contain either a **–p** or **–g** option, the value of the **STOP_ALL** state variable lets the CLI know if it should stop processes or groups.

dfocus {p2 p3 p4} dwatch –p *p
Watches the address pointed to by **p** in processes 2, 3, and 4. The **–p** option indicates that only the process triggering the watchpoint is stopped.

wa * aString –length 30 –e { goto $447 }
Watches 30 bytes of data beginning at the location pointed to by **aString**. If any of these bytes change, execution control transfers to line 447.

dwhat

Determines what a name refers to

Format:

dwhat *symbol-name*

Arguments:

symbol-name	Fully or partially qualified name specifying a variable, procedure, or other source code symbol.

Command alias:

wh

Description:

The **dwhat** command tells the CLI to display information about a named entity within a program. The displayed information contains the name of the entity and a description of the name. The examples that follow show many of the kinds of elements that this command can display.

NOTE To view information on CLI state variables or user-defined commands, you need to use the *dset* or *alias* commands.

The target focus constrains the query to a particular context.

The default width for this command is thread (**t**).

Examples:

dprint timeout

```
timeout = {
    tv_sec = 0xc0089540 (-1073179328)
    tv_usec = 0x000003ff (1023)
}
```

dwhat timeout

```
In thread 1.1:
Name: timeout; Type: struct timeval; Size: 8 bytes;
Addr: 0x11fffefc0
    Scope: #fork_loop.cxx#snore \
        (Scope class: Any)
    Address class: auto_var (Local variable)
```

wh timeval

```
In process 1:
Type name: struct timeval; Size: 8 bytes; \
        Category: Structure
    Fields in type:
    {
    tv_sec      time_t    (32 bits)
    tv_usec     int       (32 bits)
    }
```

dlist

```
20    float field3_float;
21    double field3_double;
22  en_check en1;
23
24  };
25
26  main ()
27  {
28    en_check vbl;
29    check_struct s_vbl;
30  >vbl = big;
31    s_vbl.field2_char = 3;
32    return (vbl + s_vbl.field2_char);
33  }
```

p vbl

```
vbl = big (0)
```

wh vbl

```
In thread 2.3:
Name: vbl; Type: enum en_check; \
        Size: 4 bytes; Addr: Register 01
    Scope: #check_structs.cxx#main \
        (Scope class: Any)
    Address class: register_var (Register variable)
```

wh en_check

```
In process 2:
Type name: enum en_check; Size: 4 bytes; \
        Category: Enumeration
    Enumerated values:
        big         = 0
        little      = 1
        fat         = 2
        thin        = 3
```

p s_vbl

```
s_vbl = {
    field1_int = 0x800164dc (-2147392292)
    field2_char = '\377' (0xff, or -1)
    field2_chars = "\003"
    <padding> = '\000' (0x00, or 0)
    field3_int = 0xc0006140 (-1073716928)
    field2_uchar = '\377' (0xff, or 255)
    <padding> = '\003' (0x03, or 3)
    <padding> = '\000' (0x00, or 0)
    <padding> = '\000' (0x00, or 0)
```

```
    field_sub = {
        field1_int = 0xc0002980 (-1073731200)
        <padding> = '\377' (0xff, or -1)
        <padding> = '\003' (0x03, or 3)
        <padding> = '\000' (0x00, or 0)
        <padding> = '\000' (0x00, or 0)
        field2_long = 0x0000000000000000 (0)
    ...
}
```

wh s_vbl

```
In thread 2.3:
Name: s_vbl; Type: struct check_struct; Size: 80 \
        bytes; Addr: 0x11ffff240
    Scope: #check_structs.cxx#main \
        (Scope class: Any)
    Address class: auto_var (Local variable)
```

wh check_struct

```
In process 2:
Type name: struct check_struct; Size: 80 bytes; \
        Category: Structure
    Fields in type:
    {
    field1_int          int             (32 bits)
    field2_char         char            (8 bits)
    field2_chars        <string>[2]     (16 bits)
    <padding>           <char>          (8 bits)
    field3_int          int             (32 bits)
    field2_uchar        unsigned char(8 bits)
    <padding>           <char>[3]       (24 bits)
    field_sub           struct sub_struct (320 bits){
        field1_int      int             (32 bits)
        <padding>       <char>[4]       (32 bits)
        field2_long     long            (64 bits)
        field2_ulong    unsigned long (64 bits)
        field3_uint     unsigned int    (32 bits)
        en1             enum en_check (32 bits)
        field3_double double            (64 bits)
    }
    ...
}
```

dwhere

Displays the current execution location and call stack

Format:

Displays locations in the call stack

dwhere [*num-levels*] [**–args**]

Displays all locations in the call stack

dwhere –a [**–args**]

Arguments:

num-levels	Restricts output to this number of levels of the call stack.
–args	Displays argument names and values in addition to program location information. If you omit this option, arguments are not shown.
–a	Shows all levels of the call stack.

Command alias:

w

Description:

The **dwhere** command prints the current execution locations and the call stacks—or sequence of procedure calls—which led to that point. Information is shown for threads in the current focus, with the default being to show information at the thread level.

Arguments control the amount of command output in two ways:

- The *num-levels* argument lets you control how many levels of the call stacks are displayed, counting from the uppermost (most recent) level; the **–all** option indicates that all levels should be shown.
- The **–args** option tells the CLI that it should also display procedure argument names and values for each stack level.

A **dwhere** command with no arguments or options displays the call stacks for all threads in the target set.

Output is generated for each thread in the target focus.

Examples:

dwhere	Displays the call stacks for all threads in the current focus.

dfocus 2.1 dwhere 1

Displays just the most recent level of the call stack corresponding to thread 1 in process 2. This shows just the immediate execution location of a thread or threads.

dfocus 2.1 dwhere –all

Displays the complete call stack for thread 1 in process 2, regardless of how many levels it includes.

w 1 –args

Displays the current execution locations (one level only) of threads in the current focus together with the names and values of any arguments that were passed into the current procedures.

f p1.< w 5

Displays the most recent five levels of the call stacks for all threads involved in process 1. If the depth of any call stack is less than five levels, all of its levels are shown.

This command is a slightly more complicated way of saying **f p1 w 5** because specifying a process width tells **dwhere** to ignore the thread indicator.

exit

Terminates the debugging session

Format:

exit [–force]

Description:

The **exit** command terminates the CLI session.

After executing this command, the CLI asks if it is all right to exit. If you answer yes, the CLI closes all TotalView processes. If you had entered the CLI from the TotalView GUI, this window is also closed.

NOTE Enter Ctrl-D to exit from the CLI window without exiting from TotalView.

The **–force** option tells the CLI that it should close all TotalView processes without asking permission.

Any processes and threads that were created by the CLI are destroyed. Any processes that existed prior to the debugging session (that is, were attached by the CLI as part of a **dattach** operation) are detached and left executing.

The **exit** operation cannot be halted by typing Ctrl-C; after it is invoked, the CLI session is over.

The **exit** and **quit** commands are interchangeable; they both do exactly the same thing.

Examples:

exit	Exits from the CLI, leaving any "attached" processes running (in the run-time environment).

help

Displays help information

Format:

help [*topic*]

Arguments:

topic	The topic or command for which the CLI prints information.

Command alias:

he

Description:

The **help** command prints information about the specified topic or command. If you do no use an argument, the CLI displays a list of the topics for which help is available.

If more than one screen of data would be displayed, the CLI fills the screen with data and then displays a *more* prompt. You can then type **Enter** to see more data or type **q** to return to the CLI prompt.

You can use the **capture** command to place help information into a variable.

Examples:

help help	Prints information describing the **help** command.

quit

Terminates the debugging session

Format:

quit [–force]

Description:

The **quit** command terminates the CLI session.

After executing this command, the CLI asks if it is all right to exit. If you answer yes, the CLI closes all TotalView processes. If you had entered the CLI from the TotalView GUI, this window is also closed.

NOTE Enter Ctrl-D to exit from the CLI window without exiting from TotalView.

The **–force** option tells the CLI that it should close all TotalView processes without asking permission.

Any processes and threads that were created by the CLI are destroyed. Any processes that existed prior to the debugging session (that is, were attached by the CLI as part of a **dattach** operation) are detached and left executing in the run-time environment.

The **quit** operation cannot be halted by typing Ctrl-C; after it is invoked, the CLI session is over.

The **quit** and **exit** commands are interchangeable; they both do exactly the same thing.

Examples:

quit	Exit the CLI, leaving any "attached" processes running (in the run-time environment).

stty

Sets terminal properties

Format:

stty [*stty-args*]

Arguments:

stty-args	One or more UNIX **stty** command arguments as defined in the **man** page for your operating system.

Description:

The CLI **stty** command executes a UNIX **stty** command on the **tty** associated with the CLI window. This lets you set all of your terminal's properties. However, this is most often used to set erase and kill characters.

If you start the CLI from a terminal using the **totalviewcli** command, the **stty** command alters this terminal's environment. Consequently, the changes you make using this command are retained within the terminal after you exit.

If you omit *stty-args*, the CLI displays information describing your current settings.

The output from this command is returned as a string.

Examples:

stty	Prints information about your terminal settings. The information printed is the same as if you had typed **stty** while interacting with a shell.
stty –a	Prints information about all of your terminal settings.
stty erase ^H	Sets the *erase* key to Backspace.
stty sane	Resets the terminal's settings to values that the shell thinks they should be. If you are having problems with command line editing, use this command. (The *sane* option is not available in all environments.)

unalias

Removes previously defined command

Format:

Removes an alias

unalias *alias-name*

Removes all aliases

unalias –all

Arguments:

alias-name	The name of the alias being deleted.
–all	Tells the CLI to remove all aliases.

Description:

The **unalias** command removes a previously defined alias. You can delete all aliases by using the **–all** option. Aliases defined in your **tvdinit.tvd** file are also deleted.

Examples:

unalias step2	Remove the **step2** alias. **step2** is now undefined and can no longer be used. If **step2** was included as part of the definition of another command, that command will no longer work correctly. However, the CLI will only display an error message when you try to execute the alias that contains this removed alias.
unalias –all	Removes all aliases.

Appendix A

CLI Command Summary

This appendix contains a summary of all CLI commands. If a command has an alias, it is displayed in parentheses after the command's name.

alias

Creates a new user-defined command

alias *alias-name defn-body*

Views previously defined commands

alias [*alias-name*]

capture

Assigns a command's output to a variable

capture *command*

dactions (ac)

Displays a list of action points

dactions [*ap-id-list*] [**–at** *source-loc*]
[**–enabled** | **–disabled**]

dassign (as)

Changes the value of a scalar variable

dassign *target value*

dattach (at)

Brings executing processes under CLI control

dattach [**-g** *gid*] [**-r** *hname*] *fname pid-list*

dbreak (b)
Creates a breakpoint at a source location

dbreak *source-loc* [**–p** | **–g**] [**–l** *lang* **–e** *expr*]

Creates a breakpoint at an address

dbreak –address *addr* [**–p** | **–g**] [**–l** *lang* **–e** *expr*]

dcont (co, CO)
Resumes execution and waits for the target process to start

dcont [**–waitany**]

ddelete (de)
Deletes the listed action points

ddelete *action-point-list*

Deletes all action points

ddelete –a

ddetach (det)
Detaches from the processes in the current focus

ddetach

ddisable (di)
Disables specific action points

ddisable *action-point-list*

Disables all action points

ddisable –a

ddown (d)
Moves down the call stack

ddown [*num-levels*]

denable (en)
Reenables specific action points

denable *action-point-list*

Reenables all disabled action points in the current focus

denable –a

dfocus (f)
Changes the target of CLI commands to this P/T set

dfocus *p/t-set*

Executes a command within this P/T Set

dfocus *p/t-set command*

dgo (g, G)
Resumes execution of target processes

dgo

dhalt (h, H)
Suspends execution of target processes

dhalt

dkill (k)
Terminates execution of target processes

dkill

dlist (l)
Displays code relative to a named location

dlist [–n *num-lines*]

Displays code relative to a named location

dlist *source-loc* [–n *num-lines*]

Displays code relative to the current execution location

dlist –e [–n *num-lines*]

dload (lo)
Loads debugging information

dload [-g *gid*] [-r *hname*] [-e] *executable*

dnext (n, N)
Steps one line, stepping over subroutines

dnext [*num-steps*]

dnexti (ni, NI)
Steps one machine instruction, stepping over subroutines

dnexti [*num-steps*]

dprint (p)
Prints the value of a variable

dprint *variable*

Prints the value of an expression

dprint *expression*

drerun (rr)
Restarts processes

drerun [*args*]

drun (r)
Starts or restarts processes

drun [*cmd_arguments*] [> *outfile*] [< *infile*]

dset
Changes a CLI state variable

dset *debugger-var value*

Views current CLI state variable(s)

dset [*debugger-var*]

dstatus (st, ST)
Shows current status of processes and threads

dstatus

dstep (s, S)
Steps lines, stepping into subfunctions

dstep [*num-steps*]

dstepi (si, SI)
Steps machine instructions, stepping into subfunctions

dstepi [*num-steps*]

dunset

Restores one CLI variable to its default value

dunset *debugger-var*

Restores all CLI variables to their default values

dunset –all

dup (u)

Moves up the call stack

dup [*num-levels*]

dwait

Blocks command input until the target processes stop

dwait [**–waitany**]

dwatch (wa)

Defines a watchpoint for a variable

dwatch *variable* [**–length** *byte-count*] [**–p** | **–g**]
[[**–l** *lang*] **–e** *expr*] [**–t** *type*]

Defines a watchpoint for an address

dwatch –address *addr* **–length** *byte-count* [**–p** | **–g**]
[[**–l** *lang*] **–e** *expr*] [**–t** *type*]

dwhat (wh)

Determines what a name refers to

dwhat *symbol-name*

dwhere (w)

Displays locations in the call stack

dwhere [*num-levels*] [**–args**]

Displays all locations in the call stack

dwhere –a [**–args**]

exit

Terminates the debugging session

exit [**–force**]

help (he)
Displays help information

help [*topic*]

quit
Terminates the debugging session

quit [**–force**]

stty
Sets terminal properties

stty [*stty-args*]

unalias
Removes an alias

unalias *alias-name*

Removes all aliases

unalias –all

Appendix B

CLI Command Aliases and Focus

This appendix lists all CLI command aliases and their default focus. The predefined uppercase alias for group-level commands is also indicated

Command	Alias	Default focus
alias	—	—
capture	—	—
dactions	ac	process
dassign	as	thread; if the current width is "process", **dassign** acts on each thread in the process
dattach	at	—
dbreak	b	Obtains focus from the setting of the SHARE_ACTION_POINT variable **true**: default to "group" **false**: default to "process"
dcont	co, CO	process
ddelete	de	process
ddetach	det	—
ddisable	di	process
ddown	d	thread; if the current width is "process", **ddown** acts on each thread in the process
denable	en	process
dfocus	f	—

Command	Alias	Default focus
dgo	g, G	process
dhalt	h, H	process
dkill	k	process; note that killing the primary process for a group always kills all of its slaves
dlist	l	thread; if the current width is "process", **dlist** iterates over all threads in the process
dload	lo	—
dnext	n, N	process When set to other width, actions are as follows: **thread**: step just the thread **process**: step thread while running process **group**: step one thread from each group member while runing all threads in the group
dnexti	ni, NI	process For other widths, see **dnext**
dprint	p	thread; if the current width is "process", **dprint** acts on each thread in the process
drerun	rr	process
drun	r	process
dset	—	—
dstatus	st, ST	thread
dstep	s, S	process For other widths, see **dnext**

Command	Alias	Default focus
dstepi	si, SI	process For other widths, see **dnext**
dunset	—	—
dup	u	thread; if the current width is "process", **dup** acts on each thread in the process
dwait	—	process
dwatch	wa	Obtains focus from the setting of the SHARE_ACTION_POINT variable **true**: default to "group" **false**: default to "process"
dwhat	wh	thread; if the current width is "process", **dwhat** acts on each thread in the process
dwhere	w	thread; if the current width is "process", **dwhere** acts on each thread in the process
exit	—	—
help	he	—
quit	—	—
stty	—	—
unalias	—	—

Glossary

ACTION POINT: A debugger feature that allows a user to request that program execution stop under certain conditions. Action points include breakpoints, watchpoints, evaluation points, and barriers.

ACTION POINT IDENTIFIER: A unique integer ID associated with an action point.

ADDRESS SPACE: A region of memory that contains code and data from a program. One or more threads can run in an address space. A process normally contains an address space.

AFFECTED P/T SET: The set of threads that will be affected by the command. For most commands, this is identical to the target p/t set, but in some cases it may include additional threads.

AGGREGATED OUTPUT: The CLI compresses output from multiple threads when they would be identical except for the p/t identifier.

ARENA: A specifier that indicates the processes, threads, and groups upon which a command executes. Arena specifiers are **p** (process), **t** (thread), **g** (group), **d** (default), and **a** (all).

AUTOMATIC PROCESS ACQUISITION: TotalView automatically detects the many processes that parallel and distributed programs run in, and attaches to them automatically so you do not have to attach to them manually. This process is called *automatic process acquisition*. If the process is on a remote machine, automatic process acquisition automatically starts the TotalView debugger server (the tvdsvr).

BARRIER: An action point specifying that processes reaching a particular location in the source code should stop and wait for other processes to catch up.

BREAKPOINT: A point in a program where execution can be suspended to permit examination and manipulation of data.

CALL STACK: A higher-level view of stack memory, interpreted in terms of source program variables and locations.

CHILD PROCESS: A process created by another process (*see* **parent process**) when that other process calls fork().

CLUSTER DEBUGGING: The action of debugging a program that is running on a cluster of hosts in a network. Typically, the hosts are homogeneous.

COMMAND HISTORY LIST: A debugger-maintained list storing copies of the most recent commands issued by the user.

CONTEXTUALLY QUALIFIED (SYMBOL): A symbol that is described in terms of its dynamic context, rather than its static scope. This includes process identifier, thread identifier, frame number, and variable or subprocedure name.

CORE FILE: A file containing the contents of memory and a list of thread registers. The operating system dumps (creates) a core file whenever a program exits because of a severe error (such as an attempt to store into an invalid address).

CORE-FILE DEBUGGING: A debugging session that examines a core file image. Commands that modify program state are not permitted in this mode.

CROSS-DEBUGGING: A special case of remote debugging where the host platform and the target platform are different types of machines.

CURRENT FRAME: The current portion of stack memory, in the sense that it contains information about the subprocedure invocation that is currently executing.

CURRENT LANGUAGE: The source code language used by the file containing the current source location.

CURRENT LIST LOCATION: The location governing what source code will be displayed in response to a list command.

DATA-SET: A set of array elements generated by TotalView and sent to the Visualizer. (See **visualizer process**.)

DBELOG LIBRARY: A library of routines for creating event points and generating event logs from within TotalView. To use event points, you must link your program with both the **dbelog** and **elog** libraries.

DBFORK LIBRARY: A library of special versions of the **fork()** and **execve()** calls used by the TotalView debugger to debug multiprocess programs. If you link your program with TotalView's **dbfork** library, TotalView will be able to automatically attach to newly spawned processes.

DEBUGGING INFORMATION: Information relating an executable to the source code from which it was generated.

DEBUGGER INITIALIZATION FILE: An optional file establishing initial settings for debugger state variables, user-defined commands, and any commands that should be executed whenever TotalView or the CLI is invoked. Must be called **.tvdrc**.

DEBUGGER PROMPT: A string printed by the CLI that indicates that it is ready to receive another user command.

DEBUGGER SERVER: *See* **tvdsvr process**.

DEBUGGER STATE: Information that TotalView or the CLI maintains in order to interpret and respond to user commands. Includes debugger modes, user-defined commands, and debugger variables.

DISTRIBUTED DEBUGGING: The action of debugging a program that is running on more than one host in a network. The hosts can be homogeneous or heterogeneous. For example, programs written with message-passing libraries such as Parallel Virtual Machine (PVM) or Parallel Macros (PARMACS) run on more than one host.

DIVE STACK: A series of nested dives that were performed in the same variable window. The number of greater than symbols (>) in the upper left-hand corner of a variable window indicates the number of nested dives on the dive

stack. Each time that you undive, TotalView pops a dive from the dive stack and decrements the number of greater than symbols shown in the variable window.

DIVING: The action of displaying more information about an item. For example, if you dive into a variable in TotalView, a window appears with more information about the variable.

EDITING CURSOR: A black rectangle that appears when a TotalView GUI field is selected for editing. You use field editor commands to move the editing cursor.

EVALUATION POINT: A point in the program where TotalView evaluates a code fragment without stopping the execution of the program.

EVENT LOG: A file containing a record of events for each process in a program.

EVENT POINT: A point in the program where TotalView writes an event to the event log for later analysis using TimeScan.

EXECUTABLE: A compiled and linked version of source files, containing a "main" entry point.

EXPRESSION: An expression consists of symbols (possibly qualified), constants, and operators, arranged in the syntax of the current source language. Not all Fortran 90, C, and C++ operators are supported.

EXTENT: The number of elements in the dimension of an array. For example, a Fortran array of integer(7,8) has an extent of 7 in one dimension (7 rows) and an extent of 8 in the other dimension (8 columns).

FIELD EDITOR: A basic text editor that is part of TotalView's interface. The field editor supports a subset of GNU Emacs commands.

FOCUS: The set of groups, processes, and threads upon which a CLI command acts. The current focus is indicated in the CLI prompt (if you are using the default prompt).

FRAME: An area in stack memory containing the information corresponding to a single invocation of a subprocedure.

FULLY QUALIFIED (SYMBOL): A symbol is fully qualified when each level of source code organization is included. For variables, those levels are executable or library, file, procedure or line number, and variable name.

GRIDGET: A dotted grid in the tag field that indicates you can set an action point on the instruction.

GROUP: When TotalView starts processes, it places related processes in families. These families are called "groups."

HOST MACHINE: The machine on which the TotalView debugger is running.

INITIAL PROCESS: The process created as part of a load operation, or that already existed in the run-time environment and was attached by TotalView or the CLI.

LVALUE: A symbol name or expression suitable for use on the left-hand side of an assignment statement in the corresponding source language. That is, the expression must be appropriate as the target of an assignment.

LHS EXPRESSION: This is a synonym for lvalue.

LOWER BOUND: The first element in the dimension of an array or the slice of an array. By default, the lower bound of an array is 0 in C and 1 in Fortran, but the lower bound can be any number, including negative numbers.

MACHINE STATE: Convention for describing the changes in memory, registers, and other machine elements as execution proceeds.

MESSAGE QUEUE: A list of messages sent and received by message-passing programs.

MPICH: MPI/Chameleon (Message Passing Interface/Chameleon) is a freely available and portable MPI implementation. MPICH was written as a collaboration between Argonne National Lab and Mississippi State University. For more information, see www.mcs.anl.gov/mpi.

MPMD (MULTIPLE PROGRAM MULTIPLE DATA) PROGRAMS: A program involving multiple executables, executed by multiple threads and processes.

MUTEX: Mutual exclusion. A collection of techniques for sharing resources so that different uses do not conflict and cause unwanted interactions.

NATIVE DEBUGGING: The action of debugging a program that is running on the same machine as TotalView.

NESTED DIVE WINDOW: A TotalView window that results from diving into an item in a variable window. A nested dive window replaces the contents of the variable window and has an undive symbol in its title bar. Diving on the undive symbol returns the original contents of the variable window.

OUT OF SCOPE: When symbol lookup is performed for a particular symbol name and it is not found in the current scope or any containing scopes, the symbol is said to be out of scope.

PARALLEL PROGRAM: A program whose execution involves multiple threads and processes.

PARCEL: The number of bytes required to hold the shortest instruction for the target architecture.

PARENT PROCESS: A process that calls **fork()** to spawn other processes (usually called child processes).

PARMACS LIBRARY: A message-passing library for creating distributed programs that was developed by the German National Research Centre for Computer Science.

PARTIALLY QUALIFIED (SYMBOL): A symbol name that includes only some of the levels of source code organization (for example, filename and procedure, but not executable). This is permitted as long as the resulting name can be associated unambiguously with a single entity.

PC: This is an abbreviation for *Program Counter.*

PROCESS: An executable that is loaded into memory and is running (or capable of running).

PROCESS GROUP: A group of processes associated with a multiprocess program. A process group includes program groups and share groups.

PROCESS/THREAD IDENTIFIER: A unique integer ID associated with a particular process and thread.

PROGRAM EVENT: A program occurrence that is being monitored by TotalView or the CLI, such as a breakpoint.

PROGRAM GROUP: A group of processes that includes the parent process and all related processes. A program group includes children that were forked (processes that share the same source code as the parent) and children that were forked with a subsequent call to **execve()** (processes that do *not* share the same source code as the parent). Contrast with **share group**.

PROGRAM STATE: A higher-level view of the machine state, where addresses, instructions, registers, and such are interpreted in terms of source program variables and statements.

P/T (PROCESS/THREAD) SET: The set of threads drawn from all threads in all processes of the target program.

PVM LIBRARY: Parallel Virtual Machine library. A message-passing library for creating distributed programs that was developed by the Oak Ridge National Laboratory and the University of Tennessee.

RVALUE: An expression suitable for inclusion on the right-hand side of an assignment statement in the corresponding source language. In other words, an expression that evaluates to a value or collection of values.

REMOTE DEBUGGING: The action of debugging a program that is running on a different machine than TotalView. The machine on which the program is running can be located many miles away from the machine on which TotalView is running.

RESUME COMMANDS: Commands that cause execution to restart from a stopped state: **dstep**, **dgo**, **dcont**, **dwait**.

RHS EXPRESSION: This is a synonym for **rvalue**.

RUNNING STATE: The state of a thread when it is executing, or at least when the CLI or TotalView has passed a request to the underlying run-time system that the thread be allowed to execute.

SERIAL LINE DEBUGGING: A form of remote debugging where TotalView and the TotalView Debugger Server communicate over a serial line.

SHARE GROUP: A group of processes that includes the parent process and any related processes that share the same source code as the parent. Contrast with **program group**.

SHARED LIBRARY: A compiled and linked set of source files that are dynamically loaded by other executables—and have no "main" entry point.

SIGNALS: Messages informing processes of asynchronous events, such as serious errors. The action the process takes in response to the signal depends on the type of signal and whether or not the program includes a signal handler routine, a routine that traps certain signals and determines appropriate actions to be taken by the program.

SINGLE STEP: The action of executing a single statement and stopping (as if at a breakpoint).

SLICE: A subsection of an array, which is expressed in terms of a lower bound, upper bound, and stride. Displaying a slice of an array can be useful when working with very large arrays, which is often the case in Fortran programs.

SOURCE FILE: Program file containing source language statements. TotalView allows you to debug FORTRAN 77, Fortran 90, Fortran 95, C, C++, and assembler.

SOURCE LOCATION: For each thread, the source code line it will execute next. This is a static location, indicating the file and line number; it does not, however, indicate which invocation of the subprocedure is involved.

SPAWNED PROCESS: The process created by a user process executing under debugger control.

SPMD (SINGLE PROGRAM MULTIPLE DATA) PROGRAMS: A program involving just one executable, executed by multiple threads and processes.

STACK: A portion of computer memory and registers used to hold information temporarily. The stack consists of a linked list of stack frames that holds return locations for called routines, routine arguments, local variables, and saved registers.

STACK FRAME: A section of the stack that contains the local variables, arguments, contents of the registers used by an individual routine, a frame

pointer pointing to the previous stack frame, and the value of the Program Counter (PC) at the time the routine was called.

STACK POINTER: A pointer to the area of memory where subprocedure arguments, return addresses, and similar information is stored.

STACK TRACE: A sequential list of each currently active routine called by a program and the frame pointer pointing to its stack frame.

STATIC (SYMBOL) SCOPE: A region of a program's source code that has a set of symbols associated with it. A scope can be nested inside another scope.

STEPPING: Advancing program execution by fixed increments, such as by source code statements.

STOP SET: A set of threads that should be stopped once an action point has been triggered.

STOPPED/HELD STATE: The state of a process whose execution has paused in such a way that another program event (for example, arrival of other threads at the same barrier) will be required before it is capable of continuing execution.

STOPPED/RUNNABLE STATE: The state of a process whose execution has been paused (for example, when a breakpoint triggered or due to some user command) but can continue executing as soon as a resume command is issued.

STOPPED STATE: The state of a process that is no longer executing, but will eventually execute again. This is subdivided into stopped/runnable and stopped/held.

STRIDE: The interval between array elements in a slice and the order in which the elements are displayed. If the stride is 1, every element between the lower bound and upper bound of the slice is displayed. If the stride is 2, every other element is displayed. If the stride is –1, every element between the upper bound and lower bound (reverse order) is displayed.

SYMBOL: Entities within program state, machine state, or debugger state.

SYMBOL LOOKUP: Process whereby TotalView consults its debugging information to discover what entity a symbol name refers to. Search starts with a particular static scope and occurs recursively, so that containing scopes are searched in an outward progression.

SYMBOL NAME: The name associated with a symbol known to TotalView (for example, function, variable, data type, and such).

SYMBOL TABLE: A table of symbolic names (such as variables or functions) used in a program and their memory locations. The symbol table is part of the executable object generated by the compiler (with the **–g** switch) and is used by debuggers to analyze the program.

TAG FIELD: The left margin in the source code pane of the TotalView process window containing boxed line numbers marking the lines of source code that actually generate executable code.

TARGET MACHINE: The machine on which the process to be debugged is running.

TARGET PROCESS SET: The target set for those occasions when operations may only be applied to entire processes, not to individual threads within a process.

TARGET PROGRAM: The executing program that is the target of debugger operations.

TARGET P/T SET: The set of processes and threads upon which a CLI command will act.

THREAD: An execution context that normally contains a set of private registers and a region of memory reserved for an execution stack. A thread runs in an address space.

THREAD EXECUTION STATE: The convention of describing the operations available for a thread, and the effects of the operation, in terms of a set of pre-defined states.

THREAD OF INTEREST: The primary thread that will be affected by a command.

TRIGGER SET: The set of threads that may trigger an action point (that is, for which the action point was defined).

TRIGGERS: The effect during execution when program operations cause an event to occur (such as, arriving at a breakpoint).

TVDSVR PROCESS: The TotalView Debugger Server process, which facilitates remote debugging by running on the same machine as the executable and communicating with TotalView over a TCP/IP port or serial line.

UNDIVING: The action of displaying the previous contents of a window, instead of the contents displayed for the current dive. To undive, you dive on the undive icon in the upper right-hand corner of the window.

UPPER BOUND: The last element in the dimension of an array or the slice of an array.

USER INTERRUPT KEY: A keystroke used to interrupt commands, most commonly defined as ^C (Ctrl-C).

VARIABLE WINDOW: A TotalView window displaying the name, address, data type, and value of a particular variable.

VISUALIZER PROCESS: A process that works with TotalView in a separate window, allowing you to see a graphical representation of program array data.

WATCHPOINT: An action point specifying that execution should stop whenever the value of a particular variable is updated.

Index

Symbols

separator character 24
$newval variable in watchpoints 94
$oldval variable in watchpoints 94
.tvdrc file 15
< first thread indicator 28
= symbol for PC of current buried stack frame 67
> symbol for PC 67
@ symbol for action point 67

A

-a switch 22
abbreviating commands 21
ac, see dactions command
action point identifiers 35, 43, 60
  never reused in a session 35
action points 35
  default for newly created 81
  deleting 55
  disabling 57
  displaying 43
  reenabling 60
actions, see dactions
adding debugger information 11
advancing and holding processes 33
advancing by steps 86
advancing program execution 33
alias command 40
aliases
  built-in 21
  commands 28
  default 40
  group 21
  group, limitations 21
  removing 105
all process specifier 29
ambiguous scoping 25
architecture 82
arena specifiers 28
  incomplete 30
  inconsistent widths 31
arenas 61, 73
  iterating over 28
ARGS variable 22, 78, 80
  modifying 22
ARGS_DEFAULT variable 22, 78, 80
  clearing 22
arguments
  command line 78, 80
  default 80
  replacing 22
as, see dassign command
assembler instructions, stepping 88
assign, see dassign command
assigning output 20
assigning output to variable 20
assigning p/t set to variable 30
associating with currently executing program 19
at, see dattach command
attach, see dattach command

B

b, see dbreak command
barrier point, defined 35
BARRIER_STOP_ALL variable 80
barriers 80
block, specifying in scope 24
blocking command input 92
blocking execution 53
blocking input 92
break, see dbreak command
breakpoints
  default file in which set 51
  defined 35, 50
  process and thread behavior at 33
  setting 8

setting at beginning of procedure 51
stopping all processes at 50
triggering 51
built-in alias 21
buried stack frame 66

C

C language escape characters 45
call stack 90
displaying 99
see also, stack frame
capture command 20, 42, 102
changing context 24
changing dynamic context 90
changing focus 61
changing frames 24
changing process thread set 27
changing program state 13
changing state 34
changing state variables 80
changing value of program variable 45
clarifying scope with dwhat 25
CLI
.tvdrc initialization file 15
actions are sequential 13
and Tcl 1, 11, 12
as source level debugger 11
command results 13
components 11
defined 1
how it operates 11
initialization 15
initialization file 15
interface 13
invoking program from shell example 16
not a library 12
output 19
-s switch 15
scoping interpretation 23
starting 14
starting from command prompt 14
starting program using 17
using within Tcl, no differences 12
CLI commands 20, 90
abbreviating 21
action points 39
alias 40
aliases 113
capture 20, 42, 102
dactions 43
dassign 45
dattach 19, 33, 47
dbreak 50
dcont 53
ddelete 55
ddetach 56
ddisable 57
ddown 24, 58
default focus 27, 28
denable 60
dfocus 61
dgo 31, 63
dhalt 64
dkill 18, 33, 65
dlist 66, 81
dload 17, 18, 33, 70
dload, returning process ID 20
dnext 72
dnexti 73
dprint 74
drerun 18, 77
drun 17, 22, 65, 78
drun, limitations 79
drun, reissuing 79
dset 22, 32, 80
dstatus 85
dstep 29, 31, 86
dstepi 88
dunset 22, 89
dup 24
dwait 92
dwatch 93
dwhat 25, 96
dwhere 31, 99
environment 37
execution control 39
exit 101
focus of 113
help 102
initialization 38
overview 37
program information 38
quit 103
stty 104
summary 107
termination 38
unalias 105
CLI variables, see state variables
closed loop, see closed loop
co, see dcont command
code, displaying 66
command aliases 28, 113
command arguments 22
clearing example 22
passing defaults 22
setting 22
command focus 113
command input, blocking 92
command line arguments 18, 78, 80
Command Line Interpreter, see CLI
command output 42
command prompts 32
default 32
format 32
setting 32
starting the CLI from 14
command summary 107
commands
assigning output to variable 20
commands, interrupting 13
commands, user-defined 40
compiler adding debugging information 11
compiler information, interpreting 23
completion rules for arena specifiers 30
components of an executing program 2
conditional watchpoints 93
cont, see dcont command

context, changing 24
continuous execution 13
control in parallel environments 33
control in serial environments 33
Control-C 13
controlling program execution 33
creating commands 40
creating new process objects 70
creating new processes 18
creating threads 63
ctrl-d to exit CLI 101, 103
current list location 58

D

d, see ddown command
dactions command 43
dassign command 45
datatype incompatibilities 45
dattach command 19, 33, 47
dbreak command 50
dcont command 53
ddelete command 55
ddetach command 56
ddisable command 57
ddown command 24, 58
de, see ddelete command
debugger
  how it operates 11
  separate from program 11
debugger initialization 15
debugger initialization file 15
  see also initialization
debugging option 11
debugging session 33
  ending 101
default aliases 40
default arguments 78, 80
  modifying 79
default focus 61
default process/thread set 27
default value of variables, restoring 89
default width specifiers 29
defining the current focus 81
delete, see ddelete command
deleting action points 55
deleting state variables 80
denable command 60
det, see ddetach command
dfocus command 61
dgo command 31, 63
dhalt command 64
di, see ddisable command
directory search paths 81
disable, see ddisable command
disabling action points 57
discarding buffered output 20
display call stack 99
displaying code 66
displaying current execution location 99
displaying error message information 82
displaying expressions 74
displaying help information 102
displaying information on a name 96
displaying values 74
dkill command 18, 33, 65
dlist command 66, 81
dload command 17, 18, 20, 33, 70
dnext command 72
dnexti command 73
down, see ddown command
dprint command 74
drerun command 18, 77
drun command 17, 22, 65, 78
  limitations 79
  reissuing 79
dset command 22, 32, 80
dstatus command 85
dstep command 29, 31, 86
dstepi command 88
dunset command 22, 89
dup command 24, 90
dwait command 92
dwatch command 93
dwhat command 25, 96
  clarifying scope 25
dwhere command 31, 99

E

effects of parallelism on debugger behavior 25
eliminating tab processing 67
en, see denable command
enable, see denable command
ending debugging session 101
error message information 82
ERROR state 82
escape characters 45
evaluating state 35
evaluation points
  defined 35
  see also dbreak
  setting 8
examining state 34
executable, specifying name in scope 24
EXECUTABLE_PATH variable 48, 67, 81
executing a start-up file 15
executing as one instruction 73
executing as one statement 72
executing assembler instructions 88
executing program, components 2
executing source lines 86
execution
  controlling 33
  halting 64
execution location, displaying 99
execution states 34
exit command 101
expression arguments 23
expression evaluation 23
expression values, printing 74

F

f, see dfocus command
file for start up 15
first thread indicator of < 28
focus
  default 61
  defining 81
  pushing 27
  restoring 27

see also dfocus command
focus of commands 113
fork_loop.tvd example program 16
frames, changing 24

G

-g option 11
g, see dgo command
go, see dgo command
group aliases 21
limitations 21
group members, stopping 80
group members, stopping flag 82
group stepping 36
group stepping behavior 87
group width specifier 29
groups, defined 26
groups, placing processes in 48

H

h, see dhalt command
halt, see dhalt command
halting execution 64
help command 102
holding and advancing processes 33
how a debugger operates 11

I

I/O redirection 78
identify process within group 36
implicitly defined process/thread set 27
incomplete arena specifier 30
inconsistent widths 31
infinite loop, see loop, infinite
INFO state 82
information on a name 96
initial process 25
initialization
.tvdrc file 15
initialization file 15, 105
typical contents 15
initialization search paths 15
initializing debugging state 15
initializing the CLI 15
input, blocking 92
instructions, stepping 88
interactive CLI 11
interface to CLI 13
interpreting compiler information 23
interrupting commands 13
invoking CLI program from shell example 16
iterating over a list 31
iterating over arenas 28

K

k, see dkill command
kill, see dkill command

L

l, see dlist command
launching processes 78
levels, moving down 58
lines for listing 81
LINES_PER_SCREEN variable 21, 81
list location 58
list, see dlist command
lists with inconsistent widths 31
lists, iterating over 31
lo, see dload command
load, see dload command
loop, infinite, see infinite loop

M

machine instructions, stepping 88
make_actions.tcl sample macro 8, 16
manager threads, running 86
matching process 87
matching thread 87
MAX_LIST variable 66, 81
mixing arena specifiers 31
more processing 20, 74
more prompt 20, 81, 102
MPMD (Multiple Program Multiple Data) 2
multiple executables 2
multiprocess program, attaching to processes 48
multiprocess programs
process groups 26

N

n, see dnext command
name, information about 96
names of symbols 22
newval variable in watchpoints 94
next, see command
nexti, see dnexti command
ni, see dnexti command
non-sequential program execution 13

O

oldval variable in watchpoints 94
omitting components in creating scope 25
omitting period in specifier 30
omitting tid 30
omitting width specifier 30
Open Command Line Window command 14
output
assigning output to variable 20
discarding 20
only last command executed returned 20
printing 19
returning 19
when not displayed 20
output from 19

P

p, see dprint command
p.t notation 28
P/T sets, see process/thread sets
parallel environments
execution control 33
parallel program, defined 25
parsing comments example 8
passing default arguments 22
pid specifier, omitting 30
print, see dprint command
printing expression values 74
printing information about current state 85
printing variable values 74

procedure, specifying name in scope 24
process numbers are unique 26
process objects, creating new 70
process width specifier 29
  omitting 30
process/set threads
  saving 30
process/thread identifier 26
process/thread notation 26
process/thread sets 27
  as arguments 27
  changing 61
  changing focus 27
  default 27
  examples 29
  implicitly defined 27
  inconsistent widths 31
  structure of 28
  target 27
  widths inconsistent 31
process_id.thread_id 29
processes
  and threads 2
  attaching to 47, 70
  creating new 18
  current status 85
  destroyed when exiting CLI 101, 103
  initial 25
  matching 87
  releasing control 56
  restarting 77, 78
  spawned 25
  starting 78
  stepping 36
  stepping behavior 87
  synchronizing 35
  terminating 18, 65
program components 1
program execution
  advancing 33
  controlling 33
program groups, defined 26
program groups, placing processes in 48
program state
  at barriers 53
  changing 13
program stepping 86
program variable, changing value 45
programs, associating with 19
PROMPT variable 32, 81
prompting when screen is full 74
PTSET variable 81
pushing focus 27

Q

qualifying symbol names 24
quit command 103
quotation marks 45

R

r, see drun command
reenabling action points 60
releasing control 56
removing aliases 105
replacing default arguments 22
replacing tabs with spaces 82
rerun, see rerun command
restarting processes 77, 78
restarting program execution 18
restoring focus 27
restoring variables to default values 89
results of entering a CLI command 13
results, assigning output to variables 20
resuming execution 33, 51, 53, 63, 65
Root window, starting CLI from 14
rr, see drerun command
rules for scoping 24
run, see drun command
running state 34

S

-s switch to CLI 15
s, see dstep command
sample programs
  make_actions.tcl 16
scope 23
scope of symbols 22
scoping as a tree 24
scoping rules 24
scoping, ambiguous 25
scoping, omitting components 25
screen size 81
scrolling output 20
search paths 81
search paths for initialization 15
separate semantics 11
sequential actions 13
set, see dset command
setting breakpoints 8
setting lines between more prompts 81
setting terminal properties 104
share groups, defined 26
SHARE_ACTION_POINT variable 81
share_in_group flag 81
shared library, specifying name in scope 24
shell, example of invoking CLI program 16
showing current status 85
si, see dstepi command
SILENT state 82
source code, displaying 66
source display location 66
source file, specifying name in scope 24
spawned process 25
SPMD (Single Program Multiple Data) 2
st, see dstatus command
stack and processes 2
stack frame 66
  moving down through 58
stack frame, see also call stack
stack movements 90
starting a process 78
starting program under CLI control 17
starting the CLI 14
start-up file 15
  tvdinit.tvd 40
state variables
  ARGS 22, 78, 80

ARGS, modifying 22
ARGS_DEFAULT 22, 78, 80
ARGS_DEFAULT, clearing 22
BARRIER_STOP_ALL 80
changing 80
deleting 80
EXECUTABLE_PATH 48, 67, 81
LINES_PER_SCREEN 21, 81
MAX_LIST 66, 81
PROMPT 32, 81
PTSET 81
SHARE_ACTION_POINT 81
STOP_ALL 82, 95
TAB_WIDTH 67, 82
TOTAL_VERSION 82
TOTALVIEW_ROOT_PATH 82
VERBOSE 82
viewing 80
state, initializing 15
static scope 90
status, see dstatus command
step, see dstep command
stepi, see dstepi command
stepping 35
stepping a group 36
stepping a process 36
stepping a thread 36
stepping behavior 86
stepping machine instructions 73, 88
stepping the target program 33
stepping, see also dnext command, dnexti command, dstep command, and dstepi command
stop, defined in a multiprocess environment 34
STOP_ALL variable 80, 82, 95
stop_group flag 82
stopped/held state 34
stopped/runnable state 34
stopping execution 64
stopping group members 80
stopping group members flag 82
stty command 104
symbol lookup 23
and context 23
symbol names 22
qualifying 24
specifying in scope 24
symbol scope 22
symbol specification, omitting components 25
symbols as arguments 23
symbols, interpreting 45
symbols, static scope 90
synchronizing processes 35
synchronous stop model 92
system variables, see state variables

T

tab processing 67
TAB_WIDTH variable 67, 82
tabs, replacing with spaces 82
target process/thread set 27, 33
target processes 64
terminating 65
target program 1
defined 2
stepping 33
Tcl
and CLI 11, 12
and the CLI 1
books for learning 2
CLI and thread lists 12
interpreter 1
version based upon 12
temporary breakpoint 87
terminal properties, setting 104
terminating debugging session 101
terminating processes 18, 65
thread numbers are unique 26
thread of interest 28, 29
thread sets, see process/thread sets
thread stepping behavior 86
thread width specifier 28
omitting 30
threads
and processes 2
creating 63
current status 85
matching 87
stepping 36
threads destroyed when exiting CLI 101, 103
tid, omitting 30
TotalView
executable 82
scoping interpretation 23
starting the CLI within 14
totalview command 15
TOTALVIEW_ROOT_PATH variable 82
TOTALVIEW_VERSION variable 82
totalviewcli command 15
triggering breakpoints 51
troubleshooting 4
tvdinit.tvd start-up file 40, 105

U

u, see dup command
unalias command 105
unconditional watchpoints 93
unique process numbers 26
unique thread numbers 26
unset, see dunset command
up, see dup command
user-defined commands 40
using quotation marks 45

V

value for newly created action points 81
values, printing 74
variables
assigning command output to 42
assigning p/t set to 30
changing values 45
obtaining addresses of 94
printing 74
setting command output to 20
watched 94
watching 93
VERBOSE variable 82
viewing state variables 80

W

w, see dwhere command
wa, see dwatch command
wait, see dwait command
WARNING state 82
watch, see dwatch command
watchpoints 93
  $newval 94
  $oldval 94
  conditional 93
  defined 35
  length of 94
  supported systems 94
wh, see dwhat command
what, see dwhat command
where, see dwhere command
width specifier 28, 30
  omitting 30